UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Diego Almeida de Oliveira

GOFTESTER: APRIMORAMENTO E IMPLEMENTAÇÃO DA BIBLIOTECA LIBFIT EM UMA FERRAMENTA PARA TESTES DE ADERÊNCIA

Florianópolis

2016

Diego Almeida de Oliveira

GOFTESTER: APRIMORAMENTO E IMPLEMENTAÇÃO DA BIBLIOTECA LIBFIT EM UMA FERRAMENTA PARA TESTES DE ADERÊNCIA

Trabalho de conclusão de curso submetido ao curso de Bacharelado em Ciência da Computação para a obtenção do Grau de Bacharel em Ciência da Computação.
Orientador: Prof. Dr. Paulo José de Freitas Filho

Florianópolis

2016

Ficha de identificação da obra elaborada pelo autor através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

A ficha de identificação é elaborada pelo próprio autor

Maiores informações em:
http://portalbu.ufsc.br/ficha

Diego Almeida de Oliveira

GOFTESTER: APRIMORAMENTO E IMPLEMENTAÇÃO DA BIBLIOTECA LIBFIT EM UMA FERRAMENTA PARA TESTES DE ADERÊNCIA

Este Trabalho de conclusão de curso foi julgado aprovado para a obtenção do Título de "Bacharel em Ciência da Computação", e aprovado em sua forma final pelo curso de Bacharelado em Ciência da Computação.

Florianópolis, 24 de outubro 2016.

Prof. Dr. Renato Cislaghi
Coordenador de Projetos

Banca Examinadora:

Prof. Dr. Paulo José de Freitas Filho
Orientador

Prof. Dr. Mauro Roisenberg

Prof. Dr. Sílvia Modesto Nassar

Dedico este trabalho aos meus pais, Jissuy e Enilce, ao meu irmão, Lucas, a minha namorada, Taíris, e todos os meus familiares e amigos que sempre me apoiaram e me fizeram chegar até aqui.

AGRADECIMENTOS

Agradeço aos meus queridos pais, Enilce e Jissuy, por toda força, carinho, compreensão, apoio e por sempre me mostrarem o melhor caminho a ser seguido. Agradeço por todo o esforço que fizeram para criar a mim e meu irmão com dignidade, respeito e educação.

Agradeço ao meu irmão Lucas pelo companheirismo, amizade e incentivo em todos os momentos de minha vida.

Agradeço meus avós Jissuy e Heloísa por terem me recebido em Florianópolis e me dado a oportunidade de crescer em diversos aspectos.

Agradeço aos meus familiares que apesar de muitos estarem distantes sempre me apoiaram, principalmente meus falecidos avós João Ribeiro e Paschoalina Benedetti, que sem eles eu dificilmente chegaria tão longe.

Agradeço à minha namorada Taíris por todo amor, carinho, atenção e por ter me ajudado nas várias fases deste trabalho e da minha graduação.

Agradeço aos meus amigos e colegas pelos momentos bons que passamos juntos e por todo apoio dado a minha jornada, tanto na UFSC quanto fora dela.

Agradeço aos meus professores por me passarem seus conhecimentos e por terem me ajudado a chegar até aqui.

Agradeço especialmente ao professor Dr. Paulo José de Freitas Filho pelas oportunidades e pela orientação para a realização deste trabalho.

Agradeço aos membros da banca Mauro Roisenberg e Sílvia Modesto Nassar, por aceitarem o convite e pelas considerações sobre este trabalho.

Muito obrigado a todas as pessoas que contribuíram diretamente ou indiretamente para que eu completasse mais essa jornada.

"Look around, choose your own ground
For long you live and high you fly
And smiles you'll give and tears you'll cry
And all you touch and all you see
Is all your life will ever be"
(Breathe - Pink Floyd, 1973)

RESUMO

Os testes de aderência são procedimentos estatísticos frequentemente empregados quando se deseja verificar se uma dada amostra aleatória de dados adere alguma distribuição teórica. Entretanto, para se aplicar os testes, todo um processo de tratamento dos dados, estimativa dos parâmetros das distribuições, gerações de tabelas de frequência, entre outros procedimentos, podem ser necessários de se realizar. Todos esses procedimentos são aplicados com o auxílio computacional, de modo a agilizar o processo e, principalmente, fornecer para o usuário uma análise dos resultados através de dados estatísticos e gráficos visualizáveis. Este trabalho concentrou-se principalmente no aperfeiçoamento da biblioteca de testes de aderência libfit, através da inclusão do teste Qui-Quadrado para algumas distribuições de probabilidade discretas, e no projeto e desenvolvimento de uma ferramenta em C++ utilizando o framework Qt. Esta ferramenta, chamada GOF-Tester, faz uso da libfit para realizar os testes de aderência e gerar resultados satisfatórios para o usuário.

Palavras-chave: Distribuição de Probabilidade, Teste de Aderência, Qui-Quadrado, libfit, C++, Qt.

ABSTRACT

Goodness-of-fit tests are tools frequently employed with the purpose of verifying whether a given sample of random data fits some theoretical probability distribution. However, in order to apply the tests, an elaborated process of manipulating samples, estimating parameters of distributions, creating frequency tables, among other procedures, might be applied. All these procedures can be accomplished with the support of computers, accelerating the entire process and providing to the user a solid analysis of the outcomes through statistic data and plotted graphs. This research focuses mostly in improving the goodness-of-fit library called libfit, by adding Qui-Square tests for some discrete distributions, and designing and implementing a goodness-of-fit tool using C++ and Qt framework. This tool, called GOFTester, uses libfit in order to perform goodness-of-fit tests and provide satisfactory results to the user.

Keywords: Probability Distribution, Goodness-of-fit Test, Chi-Square, libfit, C++, Qt.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

A-D	Teste Anderson Darling
DP	Distribuição de Probabilidade
FDA	Função Distribuição Acumulada
FDP	Função Densidade de Probabilidade
K-S	Teste Kolmogorov-Smirnov
M&S	*Modeling and Simulation*
MLE	*Maximum Likelihood Estimation*
UI	*User Interface*

LISTA DE SÍMBOLOS

a	Parâmetro DP Uniforme Discreta, Uniforme e Triangular
α	Parâmetro de forma e nível de significância
A_n^2	Estatística do teste Anderson-Darling
b	Parâmetro DP Uniforme Discreta, Uniforme e Triangular
β	Parâmetro DP Exponencial, Gamma, Weibull e Beta
c	Moda da DP Triangular
D_n	Estatística do teste Kolmogorov-Smirnov
fe_j	Frequência esperada em uma classe ou intervalo j
fo_j	Frequência observada em uma classe ou intervalo j
$F_n(x)$	Função de distribuição empírica
$F(x)$	Função distribuição acumulada
$f(x)$	Função densidade
FMP	Função massa de probabilidade
H_0	Hipótese nula de teste de hipótese
H_1	Hipótese alternativa de teste de hipótese
λ	Parâmetro DP Poisson
μ	Média amostral
n	Tamanho de uma amostra de dados
χ^2	Estatística do teste Qui-Quadrado
S	Desvio padrão de uma amostra
$valor\_p$	Probabilidade de significância
$\bar{X}$	Média aritmética de uma amostra

LISTA DE LISTAGENS

SUMÁRIO

1 INTRODUÇÃO

A sociedade atual, em contraste com civilizações mais antigas, já chegou a um patamar em que muitas conquistas foram atingidas e grandes feitos já foram realizados, como por exemplo a invenção dos aviões, que hoje carregam centenas de pessoas. Porém, junto a essa evolução, desafios maiores e mais complexos vão surgindo, como no caso dos aviões, no qual um grande desafio era colocá-los no ar com algum piloto que já possuísse alguma experiência de voo sem nunca ter pilotado antes. Quando alguém se depara com problemas desse tipo e tenta contorná-los, diversos fatores podem dificultá-los e torná-los inviáveis ou até mesmo impossíveis de serem tratados diretamente.

Para auxiliar nesta tarefa, um dos caminhos é o uso do paradigma de Modelagem e Simulação de sistemas, ou *Modeling and Simulation* (M&S). A simulação de sistemas, como definido por Shannon (1998), é uma técnica usada para projetar um modelo de um sistema real e fazer experimentos com este modelo, permitindo então uma melhor compreensão do comportamento do sistema e a possibilidade de avaliar suas possíveis estratégias de operabilidade. M&S pode ser usado em, por exemplo, modelos para:

- Um sistema de pilotagem de avião para que se possa fazer avaliações e otimizações do mesmo, ou treinar futuros pilotos;
- Simular a quantidade de clientes por hora nos caixas de um supermercado, de forma a se saber em quais horários é necessário ter mais operadores de caixas disponíveis;
- Um sistema de previsão de tempo.

Diversas variáveis aleatórias podem estar envolvidas no sistema real em questão, cada uma apresentando uma natureza não determinística distinta da outra. Para representá-las de forma fiel e confiável, pode-se obter uma amostragem de dados que caracterize esse comportamento. O próximo passo é utilizar modelos estatísticos que representem bem essas variáveis aleatórias. A escolha desse modelo pode ser realizada com o auxílio da estimativa por máxima verossimilhança, que é um método para se estimar os parâmetros de algum modelo estatístico, como por exemplo estimar a média e o desvio padrão de uma distribuição Normal.

Entretanto, apesar da obtenção dos parâmetros de uma distribuição de probabilidade (DP) não ser um procedimento complexo em

alguns casos, ainda é preciso medir o quão bem um conjunto de dados segue uma determinada distribuição, isto é, se adere ou não a aquela DP. Atualmente já existem métodos que realizam essa avaliação, que são conhecidos como testes de aderência.

Existem diversos tipos de testes de aderência na literatura, sendo que os testes Anderson-Darling, Qui-Quadrado e Kolmogorov-Smirnov estão entre os mais conhecidos. Um dos focos deste trabalho é dar continuidade a biblioteca libfit (FORMIGHIERI, 2007), que tem como principal objetivo a realização de testes de aderência. A biblioteca, entretanto, fornece somente testes Qui-Quadrado para distribuições contínuas. Por este motivo, um dos objetivos é aprimorar a biblioteca com a inclusão do Qui-Quadrado para distribuições discretas.

Como apontado por Freitas Filho (2008), uma das maneiras de avaliar o resultado de um teste de aderência é através da forma visual fazendo comparação de gráficos. Ao comparar a distância entre um histograma gerado através da amostra de dados e o gráfico da distribuição teórica, isto é, o gráfico da distribuição aderida, quanto menor a distância melhor é a aderência entre as duas. Baseado na avaliação visual e teórica (estatística), o outro foco deste trabalho é no projeto e implementação de uma ferramenta para realizar testes de aderência utilizando a biblioteca libfit, de forma que possua uma interface amigável no qual o usuário possa, principalmente, avaliar os resultados dos testes através de resultados estatísticos e com a visualização de gráficos.

1.1 JUSTIFICATIVA

Os testes de aderência são procedimentos estatísticos que já foram bastante estudados na literatura. Além de existir uma infinidade de estudos realizados, também existem muitas ferramentas desenvolvidas que proveem a realização desses testes. Algumas dessas ferramentas são acessíveis aos usuários comuns, como os estudantes, enquanto que outras já não são tão acessíveis. Além do fato delas muitas vezes terem custos elevados, geralmente são de propósito mais geral, como no caso de alguns softwares para se realizar modelagem e simulação de sistemas.

Se um usuário deseja realizar somente testes de aderência, ele acaba se encontrando em uma situação no qual é preciso obter softwares mais robustos que possuem testes estatísticos acoplados ao seu sistema. A opção de se obter uma ferramenta gratuita que realiza os testes, no qual o usuário pode comparar resultados através de dados estatísticos ou de forma visual com gráficos, mostra-se uma boa solução para esse

problema.

Essa ferramenta, cujo nome atribuído é GOFTester, foi projetada e desenvolvida de forma a usar a biblioteca libfit para realizar os testes de aderência. A biblioteca, graças a sua modularidade, pode ser acoplada ao sistema de forma que se possa reaproveitar todas suas funções implementadas, sem a necessidade de reimplementar funções de testes de aderência. Assim, pode-se notar que essa portabilidade permite que qualquer desenvolvedor reutilize o código implementado na libfit em algum programa que ele esteja desenvolvendo, sem a necessidade de que o programa fique dependente de outras ferramentas.

1.2 OBJETIVOS

Nesta seção são apresentados os objetivos gerais e específicos deste trabalho.

1.2.1 Objetivo Geral

O objetivo deste trabalho é aprimorar a biblioteca libfit e implementar em uma ferramenta para testes de aderência.

1.2.2 Objetivos Específicos

1. Identificar quais distribuições não estão presentes na libfit para realizar o teste Qui-Quadrado, e definir quais serão implementadas.
2. Adicionar à libfit as distribuições selecionadas para serem implementadas.
3. Projetar e implementar uma ferramenta que faça uso da libfit para realizar testes de aderência.
4. Avaliar esta ferramenta.

1.3 METODOLOGIA

Para atingir os objetivos deste trabalho, as seguintes etapas são realizadas:

Etapa 1: Elaboração da fundamentação teórica

Com o objetivo de garantir uma sólida compreensão da problemática, de modo que se possa dar inicio a modelagem da solução, primeiro é feito um estudo da fundamentação teórica que abrange a temática deste trabalho. Assuntos como distribuições de probabilidades, estimativa por máxima verossimilhança e testes de aderência são abordados.

Etapa 2: Revisão do estado da arte

É feita uma avaliação de algumas ferramentas já existentes que aplicam testes de aderência. Deste modo é possível realizar uma comparação entre as funcionalidades existentes em cada uma delas, e então coletar possíveis requisitos para a modelagem proposta neste trabalho.

Etapa 3: Aprimoramento da biblioteca libfit

Consiste nas atividades:

1. Compreensão das características da libfit, por exemplo sua arquitetura (em termos de uma biblioteca em C++), classes e fluxo de execução.
2. Modelagem do aprimoramento da libfit. Com base nas características da biblioteca, é abordada a maneira pelo qual a solução, isto é, a inclusão de novas funcionalidades será realizada.
3. Implementação das novas funcionalidades. Trata de aspectos relevantes com respeito a implementação.

Etapa 4: Projeto da ferramenta GOFTester

Modelagem da ferramenta de testes de aderência, onde são realizados procedimentos de engenharia de software, como análise de requisitos, casos de uso e diagrama de classes de projeto.

Etapa 5: Implementação da ferramenta GOFTester

Detalhes da implementação da ferramenta são apresentados, como algumas classes e trechos de códigos mais relevantes.

Etapa 6: Avaliação da ferramenta

Avaliação daquilo que se conseguiu obter como resultado deste trabalho. Comparações entre a ferramenta GOFTester e as apresentadas no estado da arte.

1.4 ESTRUTURA DO TRABALHO

Este trabalho está dividido em 5 capítulos, onde são abordadas as etapas descritas na seção 1.3.

No capítulo 2 é feito um estudo da fundamentação teórica, onde são apresentados os diferentes tipos de distribuições de probabilidades e os relacionamentos entre elas. Em seguida, expõe-se questões sobre a estimação de parâmetros através de MLE. Por fim são apresentados alguns testes de aderência, com destaque para o Qui-Quadrado.

No capítulo 3 é realizada uma análise do estado da arte, pesquisando e apresentando outros programas que realizam testes de aderência.

No capítulo 4 é abordada a proposta deste trabalho. É feito todo um processo de modelagem das soluções e desenvolvimento tanto das melhorias da libfit quanto da ferramenta GOFTester.

Por fim, no capítulo 5 trata das considerações finais, como conclusão, objetivos atingidos e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os principais temas envolvidos neste trabalho. Primeiramente, são expostas as distribuições de probabilidade implementadas e as que já se encontravam presentes na libfit. Em seguida, antes de abordar os testes de aderência, é feita uma breve explicação sobre a obtenção dos parâmetros das distribuições através de estimadores de Máxima Verossimilhança. Finalmente, são abordados os testes de hipóteses, sendo os testes de aderência apresentados de forma mais detalhada com ênfase no Qui-Quadrado.

2.1 DISTRIBUIÇÕES DE PROBABILIDADE

Existem inúmeras classes de distribuições de probabilidades, cada uma apresentando uma característica que se mostra útil para uma diferente aplicação. Certas distribuições possuem propriedades que têm se mostrado bastante úteis em simulações de sistemas, sendo assim mais relevantes para este trabalho.

A seguir são exploradas de forma breve as distribuições que já estão presentes na libfit, os quais são usadas para verificar se uma amostra de dados segue alguma delas. Além destas, que são todas contínuas, também são apresentadas algumas distribuições discretas implementadas neste trabalho. Para ambas contínuas e discretas serão exibidas as funções de distribuições acumuladas (FDA), que são utilizadas para realizar os testes de aderência e estimação de parâmetros das distribuições (FORMIGHIERI, 2007; TENÓRIO, 2005). Para as distribuições contínuas serão também expostas as funções densidades de probabilidade $f(x)$, ou FDP, e para as discretas as funções massa de probabilidade $p(x)$, ou FMP. Na seção 2.1.3 discute-se brevemente os relacionamentos entre as distribuições e, posteriormente, na seção 2.2 são discutidas algumas formas de se estimar os parâmetros de cada DP.

Como apresentado por Freitas Filho (2008), Banks et al. (2005) e Law (2015), as distribuições de probabilidade possuem diversas propriedades e aplicações, como é apresentado nas seções seguintes.

2.1.1 Distribuições Discretas

Distribuições de probabilidade discretas são aquelas cujas variáveis podem ter um número finito de valores. Tais distribuições normalmente representam fenômenos como o número de ocorrências de algum evento, por exemplo o lançamento de dados, ou a quantidade de eventos que ocorrem em um intervalo de tempo. Existem diversas distribuições discretas, com diferentes aplicações, como as distribuições Binomial, Geométrica, Bernoulli, etc. Neste trabalho serão abordadas as distribuições Uniforme Discreta e a Poisson, sendo ambas bastante usadas em M&S, principalmente a Poisson, utilizada por exemplo na Teoria de Filas (FREITAS FILHO, 2008).

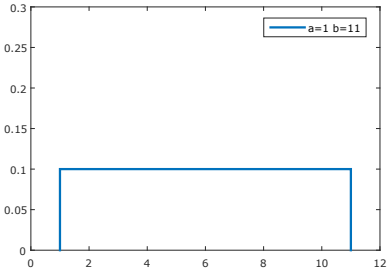
2.1.1.1 Uniforme Discreta

A Uniforme Discreta é uma DP que é bastante utilizada quando os valores aleatórios são inteiros que têm a mesma probabilidade de acontecer, como por exemplo a probabilidade de se obter um número específico no lançamento de um dado. É determinada pelos dois parâmetros a e b, que definem o valor mínimo e máximo do intervalo, respectivamente, sendo a o parâmetro de localização e $b - a$ o de escala.

$$p(x) = \begin{cases} \frac{1}{b-a+1} & \text{se } x \in \{a, a+1, ..., b\} \\ 0 & \text{caso contrário} \end{cases}$$

$$F(x) = \begin{cases} 0 & \text{se } x < 1 \\ \frac{\lfloor x \rfloor - a + 1}{b-i+1} & \text{se } a \leq x \leq b \\ 1 & \text{se } b < x \end{cases}$$

Figura 1: Função massa da distribuição Uniforme Discreta



Fonte: Elaborada pelo autor.

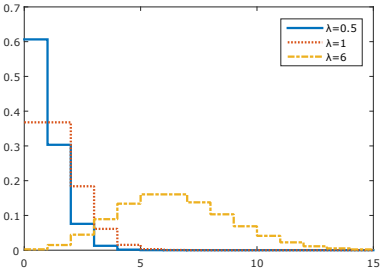
2.1.1.2 Poisson

De grande importância em M&S, a distribuição de Poisson é muito utilizada para representar a quantidade de eventos que ocorrem em um intervalo de tempo. É uma DP simples que possui o parâmetro λ.

$$p(x) = \begin{cases} \frac{e^{-\lambda}\lambda^x}{x!} & \text{se } x \in \{0, 1, ...\} \\ e^{-\lambda} \sum_{i=0}^{\lfloor x \rfloor} \frac{\lambda^i}{i!} & \text{se } 0 \leq x \end{cases}$$

$$F(x) = \begin{cases} 0 & \text{se } x < 0 \\ e^{-\lambda} \sum_{i=0}^{\lfloor x \rfloor} \frac{\lambda^i}{i!} & \text{se } 0 \leq x \end{cases}$$

Figura 2: Função massa da distribuição Poisson



Fonte: Elaborada pelo autor.

2.1.2 Distribuições Contínuas

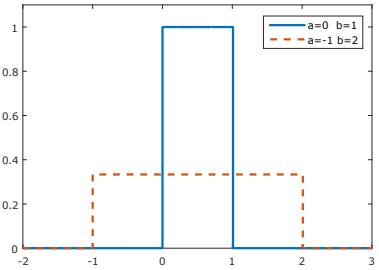
2.1.2.1 Uniforme

Essa é a DP mais simples, mas que é bastante usada em M&S na geração de eventos aleatórios, graças a sua distribuição uniforme de valores. É também utilizada em situações quando não se tem muita informação de alguma variável aleatória de algum sistema que está sendo simulado, com exceção dos valores mínimo e máximo. Os parâmetros que a definem são dois: o a, que é o parâmetro de localização, assim como o menor valor do intervalo definido pela distribuição; e o b, que é o parâmetro de escala e o maior valor do intervalo.

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{se } a \leq x \leq b \\ 0 & \text{caso contrário} \end{cases}$$

$$F(x) = \begin{cases} 0 & \text{se } x < a \\ \frac{x-a}{b-a} & \text{se } a \leq x \leq b \\ 1 & \text{if } b < x \end{cases}$$

Figura 3: Funções densidades da distribuição Uniforme



Fonte: Elaborada pelo autor.

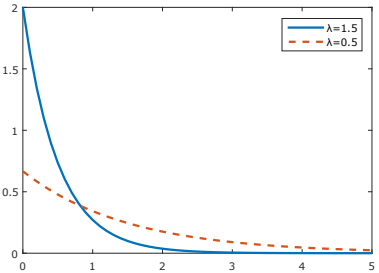
2.1.2.2 Exponencial

Distribuição que geralmente é usada para representar o tempo entre dois eventos que apresentem uma forte aleatoriedade. É determinada pelo parâmetro de escala β.

$$f(x) = \begin{cases} \frac{1}{\beta} e^{-x/\beta} & \text{se } x \geq 0 \\ 0 & \text{caso contrário} \end{cases}$$

$$F(x) = \begin{cases} 1 - e^{-x/\beta} & \text{se } x \geq 0 \\ 0 & \text{caso contrário} \end{cases}$$

Figura 4: Funções densidades da distribuição Exponencial



Fonte: Elaborada pelo autor.

2.1.2.3 Gamma

A distribuição Gamma pode ser considerada uma generalização da distribuição Exponencial. Ela geralmente é utilizada como um somatório de distribuições exponenciais independentes que representam um fenômeno aleatório observado em etapas. É bastante semelhante a distribuição Erlang, com a diferença de que a quantidade de distribuições exponenciais da Erlang, no caso da Gamma, pode ser um número não inteiro (FREITAS FILHO, 2008). Esse número é o parâmetro de forma, α, enquanto que β é o parâmetro de escala, que é a média das distribuições exponenciais. Ambos valores são os parâmetros da distribuição Gamma.

$$f(x) = \begin{cases} \frac{\beta^{-\alpha} x^{\alpha-1} e^{-x/\beta}}{\Gamma(\alpha)} & \text{se } x > 0 \\ 0 & \text{caso contrário} \end{cases}$$

$$F(x) = \begin{cases} \Gamma_p(\alpha, x/\beta) & \text{se } x > 0 \\ 0 & \text{caso contrário} \end{cases}$$

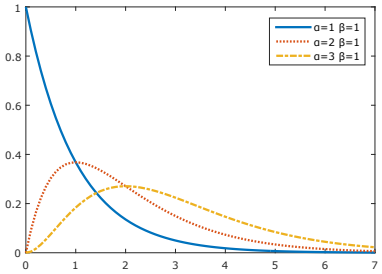
Sendo que a função Γ_p, chamada de função Gamma Incompleta, é obtida da seguinte forma:

$$\Gamma_p = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{\alpha-1} dt \qquad (a > 0)$$

E a função $\Gamma(a)$, que é a função Gamma, é definida como:

$$\Gamma(a) = \int_0^\infty t^{a-1} e^{-t} dt$$

Figura 5: Funções densidades da distribuição Gamma



Fonte: Elaborada pelo autor.

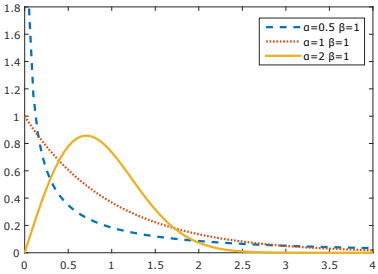
2.1.2.4 Weibull

A Weibull é uma distribuição frequentemente usada por engenheiros para representar uma variável aleatória de algum sistema ou equipamento. Normalmente essas variáveis representam algum componente crítico de um sistema que represente o tempo para ocorrer alguma falha no mesmo. Essa distribuição é composta por dois parâmetros, o α e o β, sendo que o α tem forte influência no formato do corpo da distribuição, fazendo com que por exemplo a função densidade da Weibull tenha o formato de curva semelhante ao da distribuição Normal, e o β é o parâmetro de escala.

$$f(x) = \begin{cases} \alpha\beta^{-\alpha}x^{\alpha-1}e^{-(x/\beta)^{\alpha}} & \text{se } x > 0 \\ 0 & \text{caso contrário} \end{cases}$$

$$F(x) = \begin{cases} 1 - e^{-(\frac{x}{\beta})^{\alpha}} & \text{se } x > 0 \\ 0 & \text{caso contrário} \end{cases}$$

Figura 6: Funções densidades da distribuição Weibull



Fonte: Elaborada pelo autor.

2.1.2.5 Normal ou Gaussiana

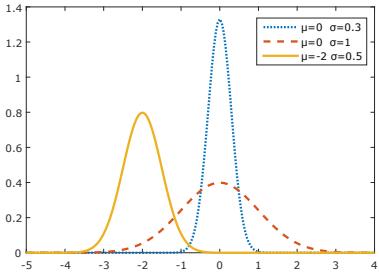
Conhecida como a curva com forma de sino, a distribuição Normal é possivelmente a mais importante entre todas as distribuições apresentadas neste trabalho. Um dos principais fatores de sua importância se da graças ao Teorema Central do Limite, o qual determina que a soma ou média de vários números aleatórios tende a formar uma distribuição Normal. Essa particularidade faz com essa distribuição seja bastante utilizada em M&S (FREITAS FILHO 2008, p. 176). Os parâmetros são o μ e o σ, que são a média e o desvio padrão, ou os parâmetros de localização e escala, respectivamente.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/(2\sigma^2)} \quad \text{para todo número real x}$$

$$F(x) = \phi(z) = \frac{1}{\sqrt{2\pi}} \int_0^z e^{-x^2/2} dx$$

Sendo $z = \frac{x-\mu}{\sigma}$.

Figura 7: Funções densidades da distribuição Normal



Fonte: Elaborada pelo autor.

2.1.2.6 Lognormal

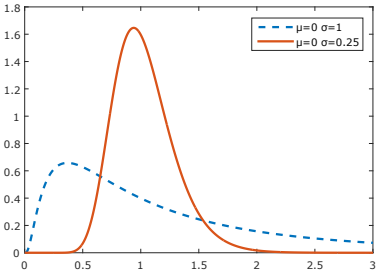
Como apontado por Law (2015), a distribuição Lognormal, ao ter seus valores aplicados a logaritmos naturais, tem o perfil de uma distribuição Normal. Esse fenômeno é normalmente explorado em variáveis aleatórias que representam tempo para completar alguma tarefa, falhas de sistemas, entre inúmeras outras aplicações. Normalmente é usada quando se tem pouca informação com respeito as amostras. Para realizar os testes de aderência ou obter os parâmetros da distribuição, ao invés de se utilizar a função de distribuição acumulada, basta aplicar o logaritmo natural nos seus valores e então tratar a Lognormal como uma Normal.

$$f(x) = \begin{cases} \frac{1}{x\sqrt{2\pi\sigma^2}} exp\frac{-(\ln x-\mu)^2}{2\sigma^2} & \text{se } x > 0 \\ 0 & \text{caso contrário} \end{cases}$$

$$F(x) = \phi\left(\frac{\ln x - \mu}{\sigma}\right)$$

Sendo ϕ a FDA da distribuição Normal.

Figura 8: Funções densidades da distribuição Lognomal



Fonte: Elaborada pelo autor.

2.1.2.7 Beta

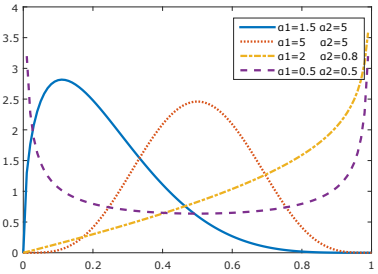
Por ser bastante simples, é uma distribuição muitas vezes usada quando não se tem dados o suficiente para utilizar uma distribuição mais robusta. Entretanto, existem algumas aplicações em que a Beta se mostra útil, como para representar a proporção de defeitos em algum lote de produtos. A distribuição Beta é definida pelos parâmetros de forma α_1 e α_2.

$$f(x) = \begin{cases} \frac{x^{\alpha_1-1}(1-x)^{\alpha_2-1}}{B(\alpha_1,\alpha_2)} & \text{se } 0 < x < 1 \\ 0 & \text{caso contrário} \end{cases}$$

$$F(x) = \frac{1}{B(\alpha_1,\alpha_2)} \int_0^x t^{\alpha_1-1}(1-t)^{\alpha_2-1} dt$$

$$B(z,w) = B(w,z) = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)} = \int_0^1 t^{z-1}(1-t)^{w-1} dt$$

Figura 9: Funções densidades da distribuição Beta



Fonte: Elaborada pelo autor.

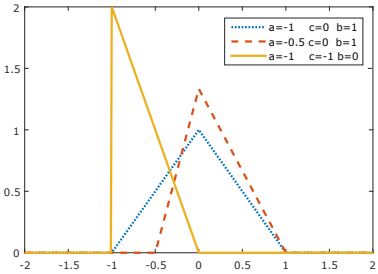
2.1.2.8 Triangular

A distribuição Triangular, devido a sua simplicidade, também é mais utilizada em situações quando não se tem muita informação. É definida pelos parâmetros a, b e c, sendo a um parâmetro de localização, $b-a$ de escala, e c, ou moda, um parâmetro de forma.

$$f(x) = \begin{cases} \frac{2(x-a)}{(b-a)(c-a)} & \text{se } a \leq x \leq c \\ \frac{2(b-x)}{(b-a)(b-c)} & \text{se } c < x \leq b \\ 0 & \text{caso contrário} \end{cases}$$

$$F(x) = \begin{cases} 0 & \text{se } x < a \\ \frac{(x-a)^2}{(b-a)(c-a)} & \text{se } a \leq x \leq c \\ 1 - \frac{(b-x)^2}{(b-a)(b-c)} & \text{se } c < x \leq b \\ 1 & \text{se } b < x \end{cases}$$

Figura 10: Funções densidades da distribuição Triangular



Fonte: Elaborada pelo autor.

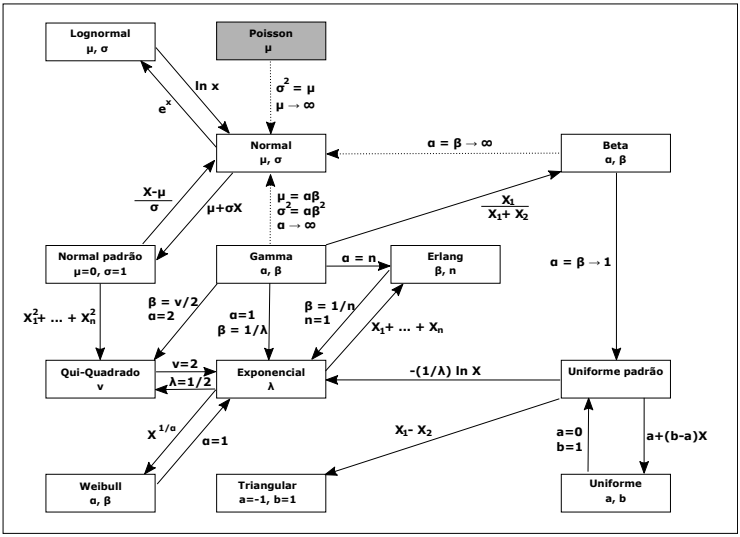
2.1.3 Relacionamento entre as Distribuições de Probabilidade

A existência de diversos tipos de distribuições de probabilidade aparenta indicar que elas são completamente diferentes entre elas. Porém, existem muitos casos em que uma determinada distribuição é de fato apenas um caso especial de alguma outra, ou uma pode ser transformada em outra a partir de uma função de transformação. Existem pelo menos três tipos de relacionamentos entre distribuições (KOKOSKA; NEVISON, 2012):

- A distribuição é um caso especial de outra;
- A distribuição pode ser transformada em outra;
- A distribuição é uma aproximação de outra. Isto significa que o limite de algum parâmetro da distribuição, tendendo a algum valor específico, aproxima ela a alguma outra distribuição (LEEMIS; MCQUESTON, 2008).

Na Figura 11, o diagrama exibe os relacionamentos entre as distribuições apresentadas neste trabalho. As flechas com linhas tracejadas representam os relacionamentos em que a distribuição é uma aproximação, enquanto que as de linha contínua representam os outros dois casos.

Figura 11: Relacionamento entre distribuições de probabilidade



Fonte: Adaptado de Kokoska e Nevison (2012).

2.2 ESTIMAÇÃO DE PARÂMETROS DE DISTRIBUIÇÕES DE PROBABILIDADE

Um dos principais propósitos deste trabalho é a aplicação dos testes de aderência para verificar se os dados obtidos pelo usuário seguem alguma distribuição teórica de probabilidades. Porém, apenas com os dados brutos não é possível fazer qualquer conclusão em relação a aderência deles. Por essa razão, a primeira etapa antes de realizar os testes é organizar os dados de modo que eles façam mais sentido e apresentem alguma forma, para que então se possa proceder com a estimativa dos parâmetros. Para realizar a primeira tarefa, pode-se organizar os dados em tabelas de frequências e histogramas, enquanto que para a obtenção dos parâmetros existem os estimadores de Máxima Verossimilhança.

A obtenção do histograma é feita através da separação dos dados

em classes de frequência. Para se gerar essas classes, uma possibilidade é criar uma tabela de frequência que filtre todos os dados da amostra em suas respectivas classes. Contudo, não existe uma regra única para definir a quantidade de classes. Uma das formas, como sugerido por Scott (1979), é usar $(1 + \log_{10} n)$ classes, sendo n o total de amostras. Outras sugestões são $\sqrt{n}$ ou $\sqrt[3]{2n}$ classes (BIRTA; ARBEZ, 2007). Porém, como indicado por Law (2015), a obtenção do número de classes é uma tarefa subjetiva, sendo então indicado que se tente diversos valores até obter um histograma com variações mais suaves. Obtido um histograma que melhor represente os dados amostrais, a próxima etapa é estimar os parâmetros das distribuições.

Tabela 1: Estimadores dos Parâmetros de Distribuições de Probabilidade Discretas e Contínuas.

Distribuição	**Parâmetro(s)**	**Estimador**
Uniforme Discreta	i, j	$i = \min_{1 \leq k \leq n} X_k, \quad j = \max_{1 \leq k \leq n} X_k$
Poisson	λ	$\lambda = \bar{X}(n)$
Uniforme	a, b	$a = \min_{1 \leq i \leq n} X_i, \quad j = \max_{1 \leq i \leq n} X_i$
Exponencial	β	$\beta = \bar{X}(n)$
Gamma	α, β	$\alpha = \frac{\bar{X}^2}{S^2}, \quad \beta = \frac{S^2}{\bar{X}}$
Weibull	α, β	α: obtido utilizando o método de Newton $\beta = \left(\frac{\sum_{i=1}^{n} X_i^{\alpha}}{n} \right)^{1/\alpha}$
Normal	μ, σ	$\mu = \bar{X}(n) \quad \sigma = \left[\frac{n-1}{n} S^2(n) \right]^{\frac{1}{2}}$
Lognormal	μ, σ	ver seção 2.1.2.6
Beta	α, β	$\alpha = \bar{X} \left[\left[\frac{\bar{X}(1-\bar{X})}{S^2} \right] - 1 \right]$ $\beta = (\bar{X} - 1) \left[\left[\frac{\bar{X}(1-\bar{X})}{S^2} \right] - 1 \right]$
Triangular	a, b	$a = \min_{1 \leq i \leq n} X_i, \quad b = \max_{1 \leq i \leq n} X_i$

Fonte: Law (2015)

A Estimação de Máxima Verossimilhança, ou *Maximum Like-*

lihood Estimation (MLE), é uma técnica usada para obter, isto é, estimar os parâmetros de algum modelo estatístico. Como mencionado por Law (2015), esse procedimento usufrui de algumas ferramentas para a estimação dos parâmetros de algumas distribuições de probabilidade, como a função de log-verossimilhança ou através de diferenciação. Porém, essas duas técnicas não são aplicáveis para todas as distribuições, sendo que em alguns casos é preciso recorrer a métodos numéricos. Na literatura já existem fórmulas e métodos para se obter os parâmetros de todas as distribuições usadas neste trabalho, como apresentado na Tabela 1.

Finalmente, feita a estimação dos parâmetros, o próximo passo é a realização dos testes de aderência para verificar se a amostra segue alguma distribuição de probabilidades.

2.3 TESTES DE HIPÓTESES

A estimação dos parâmetros é uma etapa crucial quando se tem como objetivo verificar se uma amostra segue alguma distribuição. Porém, é bastante improvável que a amostra siga perfeitamente a distribuição teórica com os parâmetro estimados. O fato de que, entre as inúmeras distribuições teóricas, a amostra irá melhor aderir possivelmente somente uma delas, é um indicativo de que, mesmo obtendo os parâmetros através de MLE, existe uma probabilidade de que a amostra siga uma outra DP ao invés da distribuição verificada.

Para se medir o quão bem uma amostra segue um determinado modelo ou distribuição, uma alternativa são os testes de aderência, que são uma classe de testes de hipóteses. De modo geral, o objetivo de um teste de hipóteses é verificar se alguma afirmação em relação a um conjunto de dados é verdadeira. Essa afirmação é chamada de hipótese nula, denotada por H_0, enquanto que a hipótese que rejeita, isto é, que contraria H_0, é denotada por H_1. Ao declarar uma afirmação, seja ela H_0 ou H_1, a ideia é criar um modelo de decisão sobre um conjunto de dados para concluir se essa afirmação é realmente verdadeira. Se for concluído que a afirmação estava errada, o que se diz é que foi um erro dizer que ela era verdadeira.

Ao se concluir que H_0 ou H_1 são verdadeiras, pode-se cair em uma situação no qual essa conclusão estava errada. Por exemplo, a hipótese nula pode ser de fato verdadeira, mas foi concluído erroneamente que ela é falsa. De acordo com Kelton, Sadowski e Sadowski (1998), esse tipo de erro é conhecido como erro do tipo I, e está as-

sociado com o nível de significância α. O nível de significância é a probabilidade de se rejeitar H_0 quando esta na verdade é verdadeira, sendo assim um falso positivo. Um outro tipo erro é o erro do tipo II, ou β, que ocorre quando H_0 é falsa mas não é rejeitada, ou seja, quando houve um falso negativo. Ambos os casos são definidos pelas equações a seguir:

$$P(H_1|H_0\ V) = P(erro\ tipo\ I) = \alpha$$

$$P(H_0|H_1\ V) = P(erro\ tipo\ II) = \beta$$

Existem vários tipos de testes aplicáveis em diversas situações. Três desses testes são apresentados neste trabalho, sendo eles o Qui-Quadrado, Kolmogorov-Smirnov e Anderson-Darling, que são testes de aderência.

2.3.1 Testes de Aderência

Os testes de aderência são testes de hipóteses em que se deseja verificar se um conjunto de dados segue (adere) alguma distribuição de probabilidade (LAW, 2015). Por se tratar de um tipo de teste de hipóteses, visa-se verificar se a hipótese nula H_0 é verdadeira. Nesse caso, a hipótese nula é a seguinte:

H_0: a amostra é composta por variáveis aleatórias com função distribuição acumulada F.

Neste trabalho são apresentados os testes Qui-Quadrado, Kolmogorov-Smirnov a Anderson-Darling. Entretanto, devido a algumas questões que dificultam a implementação dos dois últimos, como explicado nas seções abaixo, somente o Qui-Quadrado é implementado. Os outros dois testes poderão ser implementados em trabalhos futuros.

2.3.1.1 Teste Qui-Quadrado

O teste Qui-Quadrado tem como primeiro passo a separação das amostras em k classes (ou intervalos), no qual para cada classe j existe uma probabilidade teórica p_j associada, isto é, um valor que representa a proporção de dados naquele intervalo referente a distribuição que se

está verificando a aderência.

Por conseguinte, para se obter a estatística do teste Qui-Quadrado χ^2, calcula-se a equação (FREITAS FILHO, 2008):

$$\chi^2 = \sum_{j=1}^{k} \frac{(fo_j - fe_j)^2}{fe_j}$$

Onde fo_j é a frequência observada na classe j, e fe_j é a frequência esperada $fe_j = np_j$, sendo n o total de amostras.

Quanto maior χ^2, pode-se se dizer que menos a amostra adere a distribuição em questão. Portanto, se $\chi^2 = 0$, então a amostra segue perfeitamente a distribuição de probabilidade.

Entretanto, como geralmente uma amostra não segue uma distribuição com $\chi^2 = 0$, o que se verifica é se a amostra segue aproximadamente a distribuição com $v = k - 1 - p$ graus de liberdade, sendo p a quantidade de parâmetros da distribuição teórica com adição de mais uma unidade.

Uma das formas de realizar essa verificação é através da obtenção do $valor\_p$, que em termos gerais indica a probabilidade em se obter com a distribuição teórica um mesmo valor obtido com a amostra, sendo que quanto menor o p, menor é essa probabilidade. Assim, define-se um nível de significância α, que é um indicador da probabilidade de se rejeitar H_0 sendo ela verdadeira, ou seja, em se cometer um erro do tipo I, e verifica-se se $valor\_p \leq \alpha$. Se verdadeira, então a hipótese nula é rejeitada.

O Qui-Quadrado é o único teste de aderência implementado na libfit. Mais detalhes sobre o teste e sua implementação são apresentados em Formighieri (2007).

2.3.1.2 Kolmogorov-Smirnov

O teste Kolmogorov-Smirnov (K-S), ao contrário do Qui-Quadrado, não precisa que as amostras sejam separadas em classes, evitando então perda de informação. O que é feito é uma comparação entre uma função de distribuição empírica $F_n(x)$ e $F(x)$ (LAW, 2015). Ainda em comparação com o Qui-Quadrado, de acordo com Bratley, Fox e Schrage (1987), o K-S tende a ser mais preciso, dado que o seu resultado é exato para qualquer tamanho n da amostra, enquanto que o Qui-Quadrado é baseado em uma aproximação que será satisfatória somente quando se tem um n grande.

O teste é realizado através do cálculo da estatística D_n, que é basicamente a maior distância vertical entre $F_n(x)$ e $F(x)$. A estatística D_n para uma amostra formada pelos valores ordenados $X_1, X_2, ...X_n$ pode ser calculada da seguinte forma:

$$D_n = max\{D_n^+, D_n^-\}$$

Sendo que,

$$D_n^+ = \max_{1\leq i\leq n}\left\{\frac{i}{n} - F(X_i)\right\}, \quad D_n^- = \max_{1\leq i\leq n}\left\{F(X_i) - \frac{i-1}{n}\right\}$$

Para verificar se a hipótese nula é verdadeira, isso pode ser feito usando uma tabela genérica (isto é, que aplicável para qualquer distribuição contínua) de valores críticos do K-S para um dado nível de significância α e n amostras. Se D_n é maior que o valor crítico obtido na tabela, então H_0 é rejeitada.

Porém, embora o K-S possui algumas vantagens em relação ao Qui-Quadrado, como já mencionado, ele apresenta também algumas desvantagens. Na prática o K-S é aplicável somente para distribuições contínuas e, além disso, o teste usando a tabela é somente consistente quando os parâmetros das distribuições são todos conhecidos, ou seja, não foram estimados. Existem algumas adaptações do K-S para algumas distribuições cujo parâmetros foram estimados, como pode ser visto em Law (2015). A tabela genérica pode ser utilizada para todas distribuições com parâmetros estimados ou não, mas o teste perderá um pouco de sua confiabilidade.

2.3.1.3 Anderson-Darling

Como já mencionado, o propósito geral do K-S é obter a maior diferença entre a distribuição acumulada empírica e a teórica. Porém, ele tem a desvantagem de não considerar os casos quando as maiores diferenças ficam na cauda da distribuição, já que o K-S fornece o mesmo peso para a diferença $|F_n(x) - F(x)|$ para qualquer x. Um outro método, desenvolvido por Anderson e Darling (1954), propõe um teste semelhante ao K-S, mas que é baseado no quadrado da diferença das distribuições acumuladas empíricas e teóricas. Esse método, chamado de teste Anderson-Darling (A-D), consegue detectar os casos das distribuições que possuem maiores diferenças na cauda, e possui uma

capacidade maior que o K-S na verificação da aderência (LAW, 2015).

A estatística A_n^2 do A-D pode ser obtida da seguinte forma:

$$A_n^2 = \left(- \left\{ \sum_{i=1}^{n} (2i-1)[\ln F(X_i) + \ln(1 - F(X_{n+1-i}))] \right\} /n \right) - n$$

Entretanto, assim como para o K-S, quando os parâmetros das distribuições foram estimados, uma adaptação da estatística A_n^2 e da tabela de valores críticos precisa ser utilizada.

Embora o A-D possua essa desvantagem, Cheng e Currie (2009) propõe um método para obter os valores críticos do A-D das distribuições que não possuem tabelas através do processo de *bootstrap*.

3 ESTADO DA ARTE

Aderência de dados são procedimentos estatísticos que podem ser encontrados em diversas ferramentas. Entre as mais conhecidas, todas têm os testes de aderência como funcionalidades adicionais dentro de um escopo de propósito um pouco mais geral, como por exemplo o Arena, que tem enfoque em M&S, ou o @Risk, que é uma add-in do Microsoft Excel voltado para análise de riscos.

Neste capítulo são apresentados alguns destes programas, sendo avaliadas questões como gratuidade, se é um add-in, tipos de testes de aderência, etc. Todas as avaliações focam somente em questões relacionadas a execução dos testes de aderência, portanto características que não tem relação aos testes não são considerados, como por exemplo alguns dados relacionados a análise de risco.

3.1 ARENA INPUT ANALYZER

O Arena Input Analyzer é um componente que acompanha o software de simulação de eventos discretos Arena. O Arena, lançado em 1982[1], é um software bastante conhecido e utilizado na área de simulação. Apesar de ser um programa pago, existe uma versão para estudantes com poucas limitações em relação a versão paga.

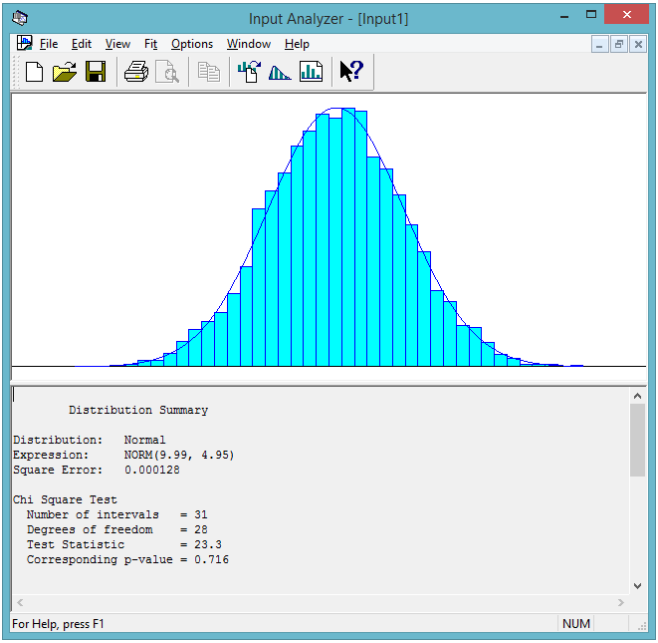
No Arena Input Analyzer, o usuário pode tanto gerar as amostras com base em alguma distribuição, como também obtê-las a partir de algum arquivo. Após a obtenção das amostras, o usuário pode escolher uma distribuição específica para aderir, ou escolher a opção "Fit All", que aplica os testes para todas as distribuições disponíveis e seleciona a que obtiver o melhor resultado.

Existem dois tipos de testes utilizados: o Qui-Quadrado e o Kolmogorov-Smirnov. Ambos são sempre aplicados nos testes de aderência.

Embora simples de usar, o Input Analyzer carece de algumas funcionalidades, como a exibição dos resultados de todas distribuições aderidas, sendo elas ordenadas pelo melhor resultado, e informações sobre os eixos do gráfico.

[1] https://pt.wikipedia.org/wiki/Rockwell_Arena/

Figura 12: *Screenshot* do Arena Input Analyzer



Fonte: Arena Input Analyzer, resultados do teste de aderência.

3.2 ORACLE CRYSTAL BALL

O Crystal Ball é um add-in do programa Microsoft Excel lançado em 1987, cujo propósito principal é sua aplicação em análise de riscos, mas que oferece também testes de aderência de dados. Não existe versão gratuita, somente uma versão de teste ou paga.

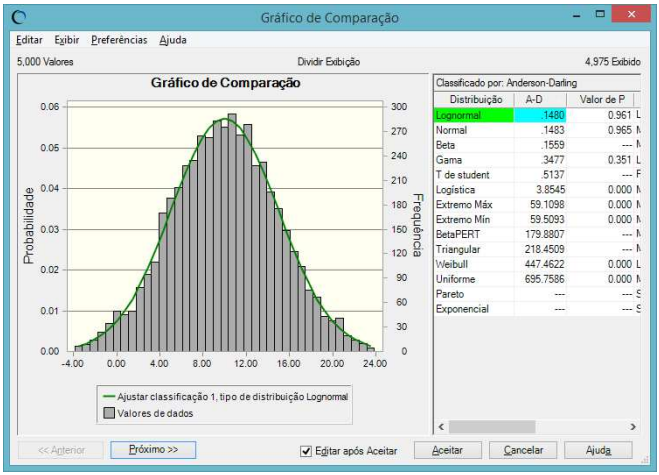
Por ser um add-in do Excel, os dados podem tanto ser obtidos a partir da interface do Excel, como também podem ser obtidos a partir da leitura de um arquivo. Quanto aos testes de aderência, o usuário pode escolher qualquer uma entre todas as distribuições para serem aderidas, assim como para os tipos de testes. Existem três testes: o Qui-Quadrado, o Kolmogorov-Smirnov e o Anderson-Darling, sendo que somente um poderá ser aplicado.

Os resultados são exibidos através da comparação gráfica entre o histograma e as curvas das funções de probabilidade das distribuições, assim como através de dados estatísticos. As distribuições são ordena-

das pelo melhor resultado, de modo que o usuário pode ir escolhendo entre uma delas para ser plotada no gráfico comparativo.

O Crystal Ball oferece uma interface de usuário bastante simples de usar e amigável. Porém, uma melhoria poderia ser a possibilidade de aplicar e exibir os resultados de mais de um tipo de teste de aderência. Outra seria poder visualizar mais de uma curva ao mesmo tempo no gráfico comparativo.

Figura 13: *Screenshot* do Oracle Crystal Ball



Fonte: Oracle Crystal Ball, resultados do teste de aderência.

3.3 @RISK

Assim como o Crystal Ball, o @Risk é um add-in do Microsoft Excel para análise de riscos, e sua primeira versão foi lançada em 1987. Não existe uma versão gratuita, somente versão de teste ou paga.

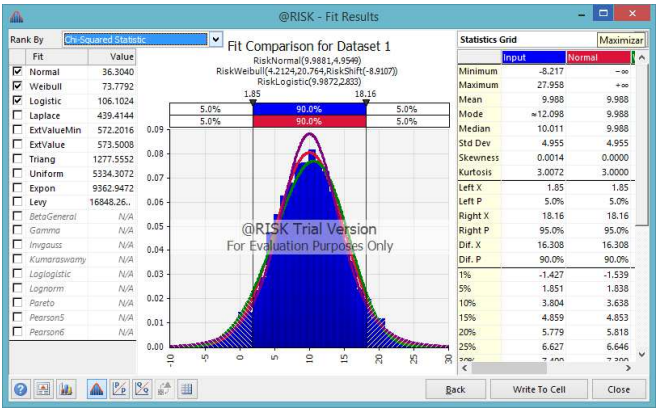
A obtenção das amostras é feita a partir da planilha do Excel, enquanto que para realizar os testes de aderência, a ferramenta oferece bastante liberdade ao usuário. É possível não somente escolher as distribuições e os tipos de testes, mas também o critério de definição dos intervalos da tabela de frequência do Qui-Quadrado, e a possibilidade de usar bootstrap. Os testes de aderência são os mesmos do Crystal Ball, sendo que todos são automaticamente aplicados.

Assim como no Crystal Ball, os resultados mostram as distribuições ordenadas de acordo com a melhor aderência, porém no @Risk

o usuário pode visualizar os resultados de qualquer tipo de teste (Qui-Quadrado, Anderson-Darling, etc). Além disto, o usuário tem a liberdade de escolher quais distribuições serão plotadas no gráfico comparativo.

Um outro diferencial é a possibilidade de aplicar bootstrap nos testes K-S e A-D, o qual, como mencionado anteriormente, pode ser utilizado para os casos em que os parâmetros das distribuições foram estimados.

Figura 14: *Screenshot* do @Risk



Fonte: @Risk, resultados do teste de aderência.

4 PROPOSTA

Neste capítulo serão apresentados os problemas e as modelagens das soluções para tanto o aprimoramento da libfit quanto para a implementação da ferramenta de testes de aderência.

4.1 APERFEIÇOAMENTO DA BIBLIOTECA LIBFIT

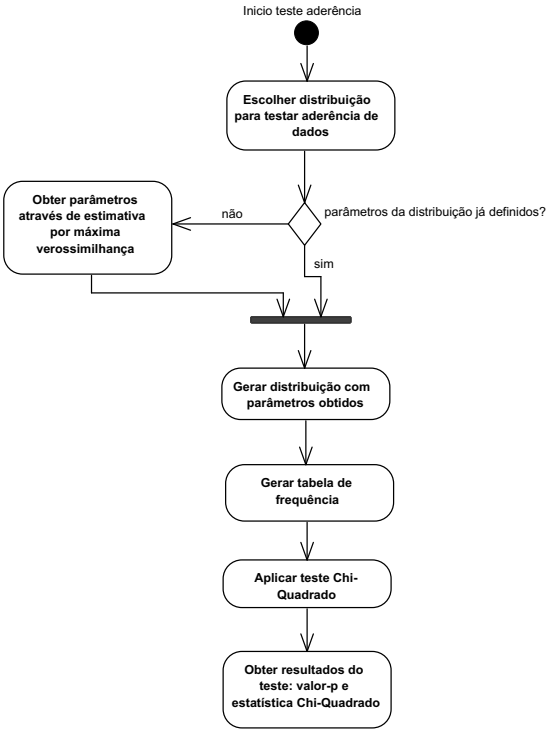
A biblioteca libfit, como já citado anteriormente, é uma ferramenta que congrega inúmeras funções estatísticas, cujo objetivo principal é poder aplicar testes de aderência de dados. Entre os variados testes existentes na literatura, o teste Qui-Quadrado foi o escolhido e implementado na primeira versão da biblioteca. Tal teste é possivelmente o mais conhecido e implementado em softwares, talvez por ser o mais antigo e mais simples de implementar. Entretanto, na libfit o Qui-Quadrado é aplicável somente em distribuições contínuas. Por esta razão este trabalho propõe a inclusão das distribuições discretas Uniforme Discreta e Poisson.

Nesta seção serão apresentadas algumas características da libfit, tais como suas classes e como elas comunicam entre elas para aplicar os testes de aderência, de forma que em seguida seja possível realizar a modelagem da solução do problema aqui proposto.

4.1.1 Características da biblioteca

Por se tratar de uma biblioteca, isto é, um conjunto de funções aplicáveis em outros programas, é desejável que ela possua uma interface simples e concisa, de forma que garanta uma intercomunicação eficiente e segura entre a aplicação e a biblioteca. Nessa interface, alguns detalhes internos não precisam (muitas vezes não devem) ser acessíveis pela aplicação, induzindo a uma abstração daquilo que é irrelevante para o programa.

Figura 15: Diagrama de atividades para teste de aderência versão original libfit

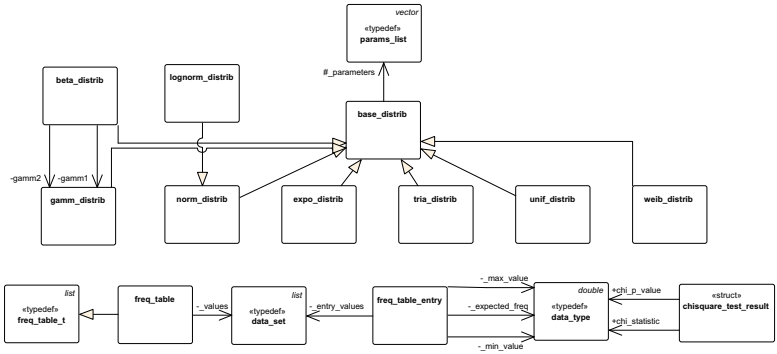


Fonte: Elaborada pelo autor.

Como a realização do teste de aderência é o principal objetivo da biblioteca, aquilo que deve ser acessível à aplicação deve não somente permitir aplicar o teste, como também fornecer dados estatísticos usados durante ou como resultado final do processo de aderência. Na Figura 15 pode-se ver as etapas mais relevantes de aplicação do teste de aderência fornecido pela libfit. Com base no fluxograma, é possível notar três etapas: obtenção da distribuição de probabilidade, geração da tabela de frequências e aplicação do teste de aderência. Toda a implementação da libfit foi feita dentro de um *namespace* chamado *fit*, cujo diagrama de classes simplificado pode ser visualizado na Fi-

gura 16. Deste diagrama pode-se notar as três principais definições que são de interesse da aplicação: as classes das distribuições, a tabela de frequência *freq_table* e os resultados do teste Qui-Quadrado presentes na estrutura *chisquare_test_result*.

Figura 16: Diagrama de classes do *namespace fit* da versão original da libfit



Fonte: Elaborada pelo autor.

Um outro elemento importante usado na libfit é o arquivo de funções utilizadas em várias etapas da biblioteca, como por exemplo para cálculo da estatística χ^2, média de uma distribuição, etc. Todas essas funções estão presentes no *namespace fit::utils*.

O que interessa para este trabalho, com relação à estrutura da biblioteca, são as distribuições de probabilidade. Todas as distribuições são classes que herdam a classe abstrata *base_distrib*, o qual possui atributos e funções em comum a todas elas, como a lista de parâmetros ou as funções de cálculo da função de probabilidade ou função acumulada. Estas funções são virtuais, o que significa que cada especialização desta classe precisa implementá-las individualmente. Não existe algo que diferencie uma distribuição contínua de uma discreta, dado que só as contínuas tinham sido implementadas.

4.1.2 Melhorias da libfit

Nesta seção é abordada a questão da inclusão das distribuições discretas Uniforme Discreta e Poisson. Como mencionado anterior-

mente, a libfit só possui distribuições contínuas, sendo que todas as funções implementadas assumem dados contínuos, isto é, que pertencem ao conjunto dos números reais. Posto isso, existem pelo menos duas abordagens para tratar desta questão:

- Criar classes abstratas para distribuições contínuas e discretas.
- Classes contínuas e discretas são tratadas de forma igual, desde que as amostras sejam adequadas para cada tipo, isto é, as discretas não podem aceitar amostras com parte decimal diferente de zero.

No primeiro caso, existe a vantagem de fornecer uma estrutura mais adequada. Porém, serão necessárias duas versões de várias funções, uma para discreta e outra para contínua, já que uma distribuição contínua, o qual possui amostras não inteiras, não pode ser tratada como uma distribuição discreta.

Quanto ao segundo caso, seria preciso saber diferenciar se a distribuição é contínua ou discreta, dependendo de onde ela estiver sendo usada. Porém, a única situação em que é necessário fazer essa diferenciação, é na criação da tabela de frequências do teste Qui-Quadrado. Seguindo esta abordagem, todas as distribuições, sejam elas contínuas ou discretas, poderão reaproveitar todo o código já implementado. Como todas as distribuições farão uso de dados do tipo *float* ou *double*, será preciso garantir que as discretas possuam amostras que, mesmo não sendo do tipo primitivo *int*, possui a parte decimal igual a zero. Neste trabalho foi decidido usar esta abordagem.

Para tratar desta abordagem, foi criado o enumerador *sample_type*, que identifica se as amostras de uma distribuição são contínuas ou discretas. Quando uma distribuição discreta for criada, as amostras passarão por um processo de validação para verificar se todos os dados são discretos. Isto pode ser realizado de forma trivial usando a função *modf* da biblioteca *cmath* para verificar se uma amostra possui parte decimal igual a zero, como pode ser visto na Listagem 4.1, cuja função é implementada no *namespace fit::utils*.

Listagem 4.1: Função de verificação do tipo das amostras

```
sample_type utils::_check_samples_type(data_set _values)
{
    double intpart;
    data_set::const_iterator it = _values.begin();
    while(it != _values.end() && std::modf((*it), &intpart)
        ↪ == 0.0)
        it++;
```

```
    if(it == _values.end())
        return sample_type::DISCRETE;
    return sample_type::CONTINUOUS;
}
```

Como indicado anteriormente, o único momento em que é necessário tratar distribuições discretas de forma diferente das contínuas é na criação da tabela de frequências do Qui-Quadrado, mais especificamente na definição da largura dos intervalos. Se as amostras são discretas, a largura precisa ter tamanho inteiro, caso contrário existiriam intervalos vazios intercalados entre os com frequências não nulas, o que afetaria o resultado do teste.

Por fim, a implementação das distribuições Uniforme Discreta e Poisson resultou nas classes *discrete_uniform_distrib* e *poisson_distrib*, cujas funções que merecem mais destaque são as seguintes:

- *cumulative_function*: Função acumulada da distribuição. É usada no teste teste de aderência no cálculo de frequência esperada.
- *probability_function*: Função de probabilidade, ou função massa de probabilidade, no caso das distribuições discretas. Pode ser utilizada para plotar curva da distribuição.
- *get_random*: Gera número aleatório de acordo com a distribuição.
- *set_params_mle*: Estima os parâmetros da distribuição usando MLE.

Todas estas funções foram implementadas baseadas em métodos fornecidos por Law (2015).

4.2 DESENVOLVIMENTO DA FERRAMENTA GOFTESTER

Com a implementação das melhorias da libfit realizada, nesta seção serão abordados o projeto e implementação da ferramenta para testes de aderência, o qual fará uso da libfit para realizar os testes. O programa GOFTester tem foco em não somente aplicar os testes de aderência, mas também na exibição dos resultados dos testes, seja ele na forma de gráficos ou de resultados com dados estatísticos.

4.2.1 Análise de Requisitos

Com o propósito de levantar todos as funcionalidades desejadas, aqui são apresentados os requisitos funcionais e não-funcionais.

4.2.1.1 Requisitos Funcionais

a) O usuário pode obter as amostras a partir de um arquivo txt ou xlsx, ou gerar aleatoriamente.

b) As amostras podem ser salvas em arquivo txt ou xlsx.

c) Amostras obtidas de arquivo precisam ser validadas.

d) Usuário pode gerar um determinado número de amostras aleatórias com base em alguma distribuição de probabilidade.

e) As amostras obtidas ou geradas devem ser exibidas ao usuário.

f) Um histograma das amostras deve ser gerado, sendo possível escolher o critério de definição das classes de frequência do histograma.

g) Deve ser possível definir o critério de definição das classes da tabela de frequências do teste Qui-Quadrado.

h) O usuário deve escolher quais distribuição serão usadas no teste para verificar aderência dos dados.

i) Particularidades de cada distribuição devem ser exibidas ao usuário, como exemplos de aplicações.

j) Distribuições intervalo do domínio é diferente do intervalo das amostras, as amostras devem ser transformadas através de algum fator de escala e *offset* para permitir a realização do teste de aderência.

k) Uma tabela com os parâmetros estimados, fator de escala e *offset* de cada distribuição deve ser exibido.

l) Os resultados dos testes de aderência de cada distribuição devem ser mostrados em uma tabela ordenada do melhor para o pior resultado.

m) Um gráfico com as curvas das funções de probabilidade das distribuições aderidas devem ser exibidas, sendo que elas devem sobrepor o histograma na mesma proporção e intervalo.

n) O gráfico pode ser exportado como arquivo de imagem.

o) Usuário pode refazer qualquer etapa da aplicação dos testes de aderência.

4.2.1.2 Requisitos Não-Funcionais

a) A implementação deve ser feita utilizando a linguagem C++ com auxílio do framework Qt.

b) A interface gráfica deve ser toda feita utilizando o Qt.

c) Deve-se utilizar a IDE Qt Creator.

d) Para realizar leitura e escrita de arquivo xlsx, deve-se utilizar a biblioteca QtXlsx.

e) A plotagem de gráficos deve ser feita utilizando a biblioteca Qwt.

f) Deve ser compatível com diferentes versões do sistema operacional Windows.

4.2.2 Projeto

Com os requisitos identificados, nesta seção são definidos os casos de uso e a modelagem da ferramenta.

4.2.2.1 Casos de Uso

- **Caso de uso:** Obter amostras de dados
  Nível: Meta do usuário
  Ator Principal: Usuário
  Stakeholders e seus interesses:
  - Usuário: Deseja obter amostras de dados para realizar testes de aderência.
  - GOFTester: Deseja acessar a biblioteca libfit para gerar amostras aleatórias.

Pós-condições:

- Sistema possui amostras válidas para gerar histograma.

Fluxo Básico:

1. Usuário escolhe uma das formas disponíveis para obter amostras.
2. Amostras são geradas ou obtidas.
3. Amostras passam por processo de validação.
4. Amostras são exibidas ao usuário.

Fluxo Alternativo:

1a. Usuário escolhe opção de importar amostras a partir de um arquivo de formato xlsx ou txt:
    1. Sistema exibe janela para usuário selecionar arquivo armazenado em algum diretório.

1b. Usuário escolhe opção de gerar amostras aleatórias de alguma distribuição de probabilidade:
    1. Sistema ativa a seleção da distribuição de probabilidade desejada e a quantidade de amostras.
    2. Sistema lista os campos para entrada de valores dos parâmetros da distribuição escolhida.
    3. Usuário insere valores dos parâmetros da distribuição e confirma.
    4. Sistema cria distribuição utilizando a biblioteca libfit.
    5. Sistema utiliza a distribuição para gerar a quantidade de amostras especificada pelo usuário.

2a. Amostras são obtidas a partir de arquivo xlsx:
    1. Se não são encontrados dados no arquivo, sistema avisa usuário.
    2. Sistema avisa que dados precisam estar na primeira coluna da primeira planilha do arquivo.

2b. Amostras são obtidas a partir de arquivo txt:
    1. Se não são encontrados dados no arquivo, sistema avisa usuário.
    2. Sistema avisa que arquivo é inválido ou algum erro foi encontrado ao efetuar leitura de dados.

2c. Amostras são geradas aleatoriamente a partir de uma distribuição de probabilidade:

1. Sistema obtém valores dos parâmetros da distribuição inseridos pelo usuário e cria distribuição utilizando a biblioteca libfit.

1a. Libfit dispara exceção de parâmetros inválidos:

1. Sistema avisa usuário quais parâmetros são inválidos.

2. Sistema utiliza a distribuição para gerar a quantidade de amostras aleatórias especificadas pelo usuário.

3a. Amostras obtidas através da leitura de arquivo xlsx ou txt são inválidas:

1. Sistema avisa usuário razão da invalidez dos dados e cancela importação de amostras.

- **Caso de uso:** Gerar histograma
  Nível: Meta do usuário
  Ator Principal: Usuário
  Stakeholders e seus interesses:

  – Usuário: Deseja visualizar um histograma de frequências das amostras em intervalos definidos por ele.

  Pré-condições:

  – Amostras válidas precisam ter sido obtidas pelo sistema.

  Pós-condições:

  – Gráfico com histograma representando as frequências das amostras em cada intervalo previamente definido.

  Fluxo Básico:

  1. Sistema exibe critérios bem definidos para definição da quantidade de intervalos do histograma
  2. Usuário escolhe critério.
  3. Sistema gera intervalos e atribui suas devidas frequências com base nas amostras.
  4. Sistema gera histograma e exibe gráfico na tela.

Repete passos 2-3 se usuário mudar critério de definição dos intervalos.

Fluxo Alternativo:

4a. Usuário altera critério dos intervalos:

1. Sistema atualiza histograma automaticamente e em tempo hábil.

- **Caso de uso:** Definir distribuições de probabilidade teóricas para verificar aderência
  Nível: Meta do usuário
  Ator Principal: Usuário
  Stakeholders e seus interesses:
  - Usuário: Deseja escolher uma ou mais distribuições, contínuas ou discretas, para realizar os testes de aderência nas amostras.
  - GOFTester: Deseja acessar a biblioteca libfit para criar distribuições de probabilidade.

  Pré-condições:
  - Histograma das amostras válidas já gerado e exibido de forma a ajudar usuário na escolha das distribuições.

  Pós-condições:
  - Todas distribuições escolhidas foram geradas e com seus parâmetros estimados a partir das amostras.

  Fluxo Básico:
  1. Sistema exibe todas as distribuições, com exemplos de suas curvas de probabilidade, além de informações como aplicações e parâmetros.
  2. Usuário escolhe as distribuições individualmente, ou escolhe opção de selecionar todas discretas, todas contínuas, ou ambas.
  3. Usuário confirma distribuições escolhidas.
  4. Sistema cria todas as distribuições selecionadas através da biblioteca libfit.

  Fluxo Alternativo:

4a. Sistema detecta que foram selecionadas distribuições discretas enquanto que amostras são contínuas:

1. Sistema avisa usuário que distribuições discretas não puderam ser criadas com os dados contínuos fornecidos.

4b. Intervalo das amostras é diferente do intervalo de alguma distribuição:

1. Sistema busca valores mínimo e máximo do intervalo da distribuição.
2. Sistema aplica fator de escala e *offset* nas amostras para deixá-las dentro de intervalo válido.
3. Valores inversos do fator de escala e *offset* são armazenados para aplicar transformação inversa dos dados após realização do teste de aderência.

4c. Libfit dispara exceção ao tentar criar uma ou mais distribuições:

1. Sistema trata exceção e exibe ao usuário quais distribuições não puderam ser criadas.

- **Caso de uso:** Aplicar teste de aderência Qui-Quadrado
  Nível: Meta do usuário
  Ator Principal: Usuário
  Stakeholders e seus interesses:
  - Usuário: Deseja aplicar teste Qui-Quadrado nas amostras com distribuições teóricas selecionadas.
  - GOFTester: Deseja acessar a biblioteca libfit para aplicar testes de aderência.

  Pré-condições:
  - Amostras obtidas e distribuições teóricas já selecionadas.

  Pós-condições:
  - Obtenção dos resultados de todos os testes aplicados.

  Fluxo Básico:

  1. Usuário seleciona critério de definição dos intervalos da tabela de frequência usada pelo Qui-Quadrado.
  2. Sistema acessa libfit para criação das tabelas de frequências de cada distribuição.

3. Sistema envia as tabelas de frequência e sua respectivas distribuições para a libfit para realizar os testes de aderência.

Fluxo Alternativo:

2-3a. Libfit dispara exceção ao tentar criar tabela de frequência e aplicar teste de aderência:

1. Sistema trata exceção e exibe ao usuário em quais distribuições o teste de aderência não pode ser aplicado.

- **Caso de uso:** Obter melhores resultados de teste de aderência
  Nível: Meta do usuário
  Ator Principal: Usuário
  Stakeholders e seus interesses:

  – Usuário: Deseja visualizar resultados dos testes de aderência, com distribuições ordenadas da que melhor aderiu para a que pior aderiu. Deseja visualizar gráfico comparativo entre histograma e distribuições aderidas.

  Pré-condições:

  – Teste de aderência já realizado e resultados obtidos para cada distribuição aderida.

  Pós-condições:

  – Relação de melhores resultados do teste de aderência de forma gráfica e com dados estatísticos.

  Fluxo Básico:

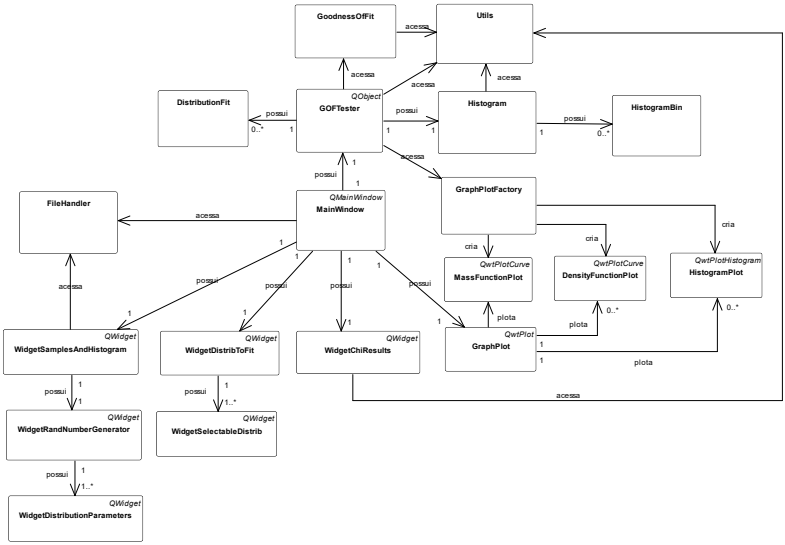
  1. Sistema ordena distribuições do melhor ao pior resultado.
  2. Sistema exibe em uma tabela os resultados ordenados de cada distribuição.
  3. Sistema exibe botões para mostrar ou esconder cada curva das distribuições. Botões são ordenados pelo melhor resultado do teste.
  4. Parâmetros estimados e fatores de escala e *offset* de cada distribuição são exibidos em uma tabela.

4.2.2.2 Modelagem da Solução

Com a elaboração dos casos de uso realizada, se tem uma visão mais detalhada do funcionamento esperado da ferramenta a ser implementada. Os casos de uso, que podem ser considerados análogos a história narradas, descrevem o passo a passo para a realização de tarefas quem atendem os requisitos levantados. Isto posto, nesse seção é apresentada uma modelagem da solução, já levando em consideração como a ferramenta será implementada de forma que reflita aquilo que foi apresentado nos casos de uso.

Como a implementação será feita em C++ usando o framework Qt, o qual apresenta características próprias que influenciam na arquitetura do programa desenvolvido, alguns detalhes precisam ser levados em consideração. O Qt é um framework baseado em *signals* e *slots*, que são propriedades que permitem que instâncias de classes diferentes possam sem comunicar sem que uma tenha conhecimento da outra. A arquitetura de um projeto desenvolvido com o Qt normalmente é separada em Visão e Modelo, diferente do padrão de projeto MVC (Model View Controller), o qual possui um controlador que faz interface entre a visão e o modelo. No caso do Qt, a visão pode exercer o papel de controle através da conexão dos componentes da visão com os do modelo, permitindo assim que eles se comuniquem através dos *signals* e *slots*. Com isso, a modelagem da solução foi feita separando os componentes em modelo e visão.

Figura 17: Diagrama de classe da ferramenta GOFTester



Fonte: Elaborada pelo autor.

O diagrama de classes do projeto pode ser visualizado na Figura 17. Apesar de ser um diagrama de classes de projeto, foram omitidas informações como atributos e funções, para efeito de simplificação e melhor entendimento das relações entre as classes. A classe MainWindow representa a janela principal do programa, todas as etapas de aplicação do teste de aderência, desde a obtenção das amostras até a exibição dos resultados dos testes, serão feitas nessa janela. Porém, cada funcionalidade, como a geração de amostras aleatórias, será atribuída a componentes de interface separados, conhecidos como Widgets.

Cada um desses componentes é uma classe que herda a classe QWidget, que é nativa do Qt. Dessa forma evita-se uma grande sobrecarga de funcionalidades que poderiam ser todas implementadas na MainWindow. Essas classes, além daquelas que herdam classes de plotagem da biblioteca Qwt (melhor explicadas mais adiante), formam a visão da arquitetura do programa. Como mencionado anteriormente, a visão exerce também o papel de controlador, sendo que a MainWindow ficará responsável pela interface entre o modelo e todos os componentes da visão.

Exercendo uma papel importante no programa, fica atribuído a

classe GOFTester a responsabilidade de gerenciar as principais funcionalidades, como obter o histograma das amostras, usar as amostras para gerar distribuições de probabilidade, realizar requisições de aplicação de teste de aderência nas distribuições e amostras, etc. Ela herda QObject para que possa se comunicar com a MainWindow através de *signals* e *slots*.

Outras classes importantes são: GoodnessOfFit, que é responsável por acessar a biblioteca libift para realizar testes de aderência; DistributionFit, que representa uma distribuição de probabilidade que, além das distribuição gerada pela libfit a partir das amostras, contém também os resultados do teste de aderência, amostras, fatores de escala e offset das amostras, etc; e Histogram, classe que contém instâncias da classe HistogramBin, que são os intervalos do histograma.

A classe GraphPlot representa o gráfico onde serão plotados itens como as curvas de funções e histogramas. Para isso ela herda a classe QwtPlot da biblioteca Qwt. As classes HistogramPlot, MassFunctionPlot e DensityFunctionPlot são os componentes gráficos que serão plotados, isto é, o histograma, a função massa de distribuições discretas e a função densidade de distribuições contínuas, respectivamente.

Por último, existem as classes de utilidades, as quais têm o papel de auxiliar em tarefas como leitura ou escrita de arquivos txt e xlsx, criação de instâncias de itens de plotagem, etc. Estas classes são as classes FileHandler e GraphPlotFactory. Além destas, a classe Utilidades conterá funções para auxiliar outras tarefas, como gerar cores variadas para plotar curvas de diferentes cores.

4.2.3 Implementação

Nesta seção são apresentados detalhes de implementação de algumas classes que têm uma maior notoriedade, com destaque para alguns trechos de código mais relevantes, além de *screenshots* das telas de UI.

4.2.3.1 Classe Histogram

A classe Histogram, como o próprio nome diz, representa um histograma, sendo este composto por intervalos que são atributos do tipo HistogramBin. HistogramBin é uma classe que contém basicamente os valores mínimo, máximo e a frequência do intervalo. Os principais atributos da classe Histogram são:

- *type_*: Se as amostras são discretas ou contínuas. Essa diferenciação é usada para definir a largura dos intervalos.
- *bins_*: Lista de intervalos, que são instâncias da classe *HistogramBin*.

Listagem 4.2: Classe Histogram: `histogram.h`

```
class Histogram
{
public:
    Histogram(const QList<double> samples, const fit::
        ↪ _bin_criteria criteria, const unsigned int
        ↪ max_bins = fit::_MAX_BINS, const int num_bins =
        ↪ 0);
    Histogram();

    // GETTERS
    ...

    // SETTERS
    ...

private:
    fit::sample_type type_;
    QList<double> samples_;
    QList<HistogramBin> bins_;
    double highest_frequency_;

    void generateFrequencies(const fit::_bin_criteria
        ↪ criteria, const unsigned int max_bins, const int
        ↪ num_bins = 0);
    void createBins(double min, double width, const int k);
    void updateBinsFrequency();
    double frequencyAt(double x);
};
```

Na Listagem 4.3 encontra-se a principal função desta classe, que é responsável por criar cada intervalo do histograma, e atribuir a eles suas devidas frequências.

A primeira etapa é o ordenamento das amostras, de maneira que otimize a atribuição de frequências dos intervalos, partindo do intervalo mais a esquerda ao mais a direita. Em seguida, é definida a quantidade de k intervalos, que depende do critério passado através do argumento *criteria*. O próximo passo é calcular a largura do intervalo. Se as amostras são discretas, a largura é arredondada para cima de modo a evitar espaços vazios entre os intervalos (embora isso possa gerar intervalos vazios nas extremidades do histograma). Por último, são geradas as instâncias de intervalos e atribuídas suas frequências.

Listagem 4.3: Classe Histogram: criação de intervalos

```
void Histogram::generateFrequencies(const fit::_bin_criteria
    ↪  criteria, const unsigned int max_bins, const int
    ↪ num_bins)
{
    // sort
    std::list<double> sorted = samples_.toStdList();
    sorted.sort();
    samples_ = QList<double>::fromStdList(sorted);

    // get min and max
    double min = samples_.front();
    double max = samples_.back();
    double variation = max - min;

    // get number of bins
    double k;
    switch (criteria) {
        case fit::_bin_criteria::SQUAREROOT:
            k = std::round(std::sqrt((double)samples_.size()
                ↪ ));
            break;
        case fit::_bin_criteria::STURGES:
            k = std::round (1.5 + 3.222 * std::log10((double
                ↪ )samples_.size()));
            break;
        case fit::_bin_criteria::OTHER:
            k = num_bins;
            break;
        default:
            k = max_bins;
    }

    k = k < max_bins ? k : max_bins;

    double width = variation / k;

    // round width up if discrete samples
    if(type_ == fit::sample_type::DISCRETE)
        width = std::ceil(width);

    width = width <= 0 ? 1 : width;

    // create bins
    createBins(min, width, (int)k);

    // update bin's frequency
    updateBinsFrequency();
}
```

4.2.3.2 Classe DistributionFit

A classe DistributionFit tem o propósito de armazenar todos os dados relacionados a uma distribuição aderida. Por essa razão, ela possui atributos como os resultados do Qui-Quadrado, tabela de frequência, entre outros. Abaixo são listados alguns atributos:

- *distribution_*: Distribuição criada pela libfit. É usada para realizar os testes de aderência aplicados pela biblioteca.
- *type_*: Tipo de distribuição (se é Exponencial, Normal, etc).
- *samples_*: Amostras ajustadas por *offset* e fator de escala para ficarem no intervalo correto dependendo do tipo de distribuição. Se amostras já estão dentro do intervalo, nenhuma transformação é aplicada.
- *offset_*: Offset usado para deslocar amostras e deixá-las no mesmo intervalo das amostras originais.
- *scale_factor_*: Fator de escala para ajustar amostras para ficarem na mesma escala das amostras originais.
- *chi_square_*: Resultados do teste Qui-Quadrado. É um par de valores que armazena a tabela de frequências e a estrutura de resultados do teste fornecidos pela libfit. A estrutura contém a estatística χ^2, o *valor_p* e o grau de liberdade.

Listagem 4.4: Classe DistributionFit: `distribution_fit.h`

```
class DistributionFit
{
public:
    DistributionFit(fit::distrib_ptr distribution, fit::
        ↪ _distribution_type type, double offset, double
        ↪ scale_factor, QList<double> samples);

    QList<QPointF> getProbabilityFunctionPoints(int n_points
        ↪ , double y_mean_hist, double width_bin, double
        ↪ n_samples);

    // GETTERS
    ...

    // SETTERS
    ...
```

```
private:
    QSharedPointer<fit::base_distrib> distribution_;
    fit::_distribution_type type_;
    QVector<QPair<QString, double>> params_;
    double min_, max_;

    // scaled data
    QList<double> samples_;
    double offset_;
    double scale_factor_;

    // fit results
    QPair<fit::chisquare_test_result, fit::freq_table>
        ↪ chi_square_;
};
```

Além dos atributos que armazenam informações importantes, principalmente sobre os resultados do teste de aderência, uma função que merece atenção é a *getProbabilityFunctionPoints*, que pode ser vista na Listagem 4.5. Essa função retorna uma lista que armazena pontos de coordenadas x e y da função de probabilidade. Como pode ter sido aplicado uma função de transformação nas amostras para que ficassem dentro do intervalo da distribuição em questão, esses pontos são ajustados para ficarem na mesma escala e intervalo do histograma, de acordo com o fator escala e *offset*. Além disso, os valores no eixo y precisam também passar por uma transformação para ficarem na mesma proporção do histograma, de modo a ser possível comparar a curva da função com o histograma plotado em um gráfico.

A transformação no eixo y é dependente do tipo de distribuição, como pode ser visto no cálculo da variável *scale_y*. Este cálculo é feito com base em heurísticas: distribuições que possuem forma semelhante a da distribuição Normal, precisam de um fator de escala baseado no total de amostras e na largura dos intervalos do histograma, enquanto que para as outras distribuições é feita uma simples transformação com base na diferença entre o y da média das amostras do histograma e da distribuição.

Listagem 4.5: Classe DistributionFit: obtenção de pontos da FDP

```
QList<QPointF> DistributionFit::getProbabilityFunctionPoints
    ↪ (int n_points, double y_mean_hist, double width_bin,
    ↪ double n_samples)
{
    // set y scale factor -> distribution dependent
    double y_mean_distrib = distribution_->
        ↪ probability_function(distribution_->get_mean());
```

```
    double scale_y = 1;
    if(type_ != fit::UNIFORM && type_ != fit::
        ↪ DISCRETE_UNIFORM && type_ != fit::EXPONENTIAL &&
        ↪ type_ != fit::TRIANGULAR && type_ != fit::
        ↪ LOGNORMAL)
        scale_y = y_mean_hist/y_mean_distrib;
    else
        scale_y = n_samples*width_bin;

    // if discrete distribution, add one point for each
        ↪ integer
    double step = (max_ - min_)/(double)n_points;
    if(distribution_->get_samples_type() == fit::sample_type
        ↪ ::DISCRETE)
    {
        step = 1;
        n_points++;
    }

    // define all scaled and offsetted points
    double x = min_;
    QList<QPointF> points;
    for(int i = 0; i < n_points; i++)
    {
        // scale and offset point
        double y_scaled = scale_y * distribution_->
            ↪ probability_function(x);
        double x_scaled = offset_ + scale_factor_*x;

        points.push_back(QPointF(x_scaled, y_scaled));

        x += step;
    }
    return points;
}
```

4.2.3.3 Classe GoodnessOfFit

Classe responsável por interagir com a biblioteca libfit para tanto realizar os testes de aderência, como também gerar números aleatórios usando um determinado tipo de distribuição. Como é uma classe cuja principal finalidade é comunicar-se com a libfit, só possui funções estáticas e nenhum atributo.

Listagem 4.6: Classe GoodnessOfFit: `goodness_of_fit.h`

```
class GoodnessOfFit
{
public:
```

```
    GoodnessOfFit();
    static QList<double> generateRandomNumbers(int
        ↪ num_samples, QVector<double> params, int
        ↪ distrib_type);
    static DistributionFit* createDistributionFit(QList<
        ↪ double> samples, fit::_distribution_type type);
    static QStringList fitDistributions(QList<
        ↪ DistributionFit*> &distributions, int
        ↪ bin_criteria);
};
```

O papel principal desta classe é a aplicação dos testes de aderência, que são realizados pela libfit. A função responsável por esta tarefa pode ser vista na Listagem 4.7. Como ela recebe todas as distribuições passíveis de se aplicar o teste de aderência, esta função só precisa acessas a libfit para criar as tabelas de frequências e aplicar os testes em cada uma das distribuições. Para aquelas que falharem, será retornada uma lista com seus nomes na forma de *string*.

Listagem 4.7: Classe GoodnessOfFit: aplicação de teste de aderência

```
QStringList GoodnessOfFit::fitDistributions(QList<
    ↪ DistributionFit *> &distributions, int bin_criteria)
{
    QStringList not_fitted_distributions; // stores
        ↪ disbritutions that failed to fit
    foreach(DistributionFit *distribution, distributions)
    {
        fit::distrib_ptr distrib_ptr = (fit::distrib_ptr)(
            ↪ distribution->getDistribution()->clone());
        fit::freq_table freq_table;
        fit::chisquare_test_result chi_result;

        try
        {
            // get frequency table
            freq_table = fit::get_freq_table(distribution->
                ↪ getSamples().toStdList(), *distrib_ptr,
                ↪ bin_criteria);

            // get chi results
            chi_result = fit::get_chisquare_test(freq_table)
                ↪ ;
        }
        catch(...)
        {
            not_fitted_distributions.push_back(distribution
                ↪ ->getName());
            distributions.removeOne(distribution);
                ↪ distributions.removeOne(distribution);
            continue;
```

```
        }
            distribution->setChiSquare(QPair<fit::
                ↪ chisquare_test_result,fit::freq_table>(
                ↪ chi_result, freq_table));
    }
    return not_fitted_distributions;
}
```

4.2.3.4 Classe GOFTester

Como já mencionado anteriormente, a classe GOFTester tem função central no programa. Ela mantém todos os elementos mais importantes, como o histograma, as distribuições, as amostras, etc. Além disto, ela é responsável por se comunicar diretamente com a visão, e efetuar solicitações às outras classes do modelo para, por exemplo, criar uma distribuição. Para poder se comunicar com a visão através de *signals*, esta classe herda a classe QObject, que é nativa do Qt, e utiliza o *macro* `Q_OBJECT`, que é também necessário para a utilização de *signals*. Segue na Listagem 4.8 alguns elementos mencionados, como os atributos e os *signals* utilizados.

Listagem 4.8: Classe GOFTester: `gof_tester.h`

```
class GOFTester: public QObject
{
    Q_OBJECT

public:
    GOFTester();

    HistogramPlot* createHistogram(const int bin_criteria,
        ↪ const int num_bins);
    QList<QPair<QString, QPair<int, double>>>
        ↪ getHistogramInfo();
    void fitDistributions(QList<int> indexes_distrib, int
        ↪ bin_criteria);

    // GETTERS
    ...

    // SETTERS
    ...

private:
    QList<double> samples_;
    Histogram histogram_;
    QList<int> indexes_distributions_;
```

```
    QList<DistributionFit*> distributions_fit_;

    QStringList createSelectedDistributions();
    void prepareAndPlotDistributions();

public slots:
    QList<double> getRandomSamples(const int numb_samples,
        ↪ const QVector<double> params, const int
        ↪ distrib_type);

signals:
    void sigPlotDistributions(QList<QwtPlotCurve*> curves);
    void sigShowDistributionsInfo(const QList<
        ↪ DistributionFit*> distributions_fit, const
        ↪ QStringList distrib_error_fit);
};
```

4.2.3.5 Classe MainWindow

Por ser a janela principal, a classe MainWindow será sempre exibida durante toda a execução do programa. Porém, diversas funcionalidades de interface do programa são atribuídas a outros componentes de interface. A medida que se evolui no processo de aplicação dos teste de aderência, diferentes componentes vão sendo exibidos. Porém, o histograma das amostras e sua tabela de frequência são sempre exibidos, independentemente da etapa do teste de aderência. Dessa forma o usuário pode sempre ficar ciente da forma do histograma e poder fazer decisões mais conscientes, como por exemplo, na escolha das distribuições a serem aderidas.

Os componentes de interface que pertencem a esta classe trabalham na forma de telas progressivas, isto é, a cada etapa do teste de aderência uma tela diferente é exibida. Cada uma dessas telas é melhor apresentada nas próximas seções.

4.2.3.6 Classe WidgetSamplesAndHistogram

Classe que exibe o componente de interface para obtenção das amostras e critério de definição dos intervalos do histograma. Existem três formas de obter as amostras:

- Leitura de dados armazenados em um arquivo excel de formato xlsx. Os dados devem estar listados na primeira coluna da planilha.

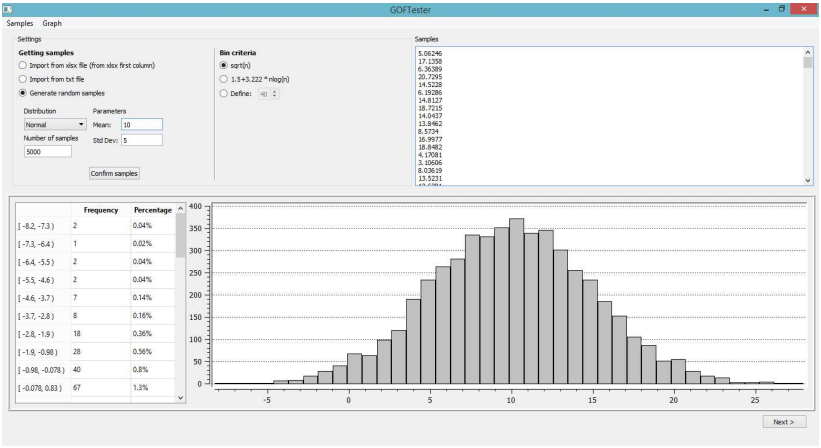
- Leitura de dados armazenados em arquivo de texto de formato txt.

- Geração de números aleatórios com base em uma distribuição de probabilidade. O usuário pode escolher uma das distribuições disponíveis e definir seus parâmetros.

Nos três casos, se alguma exceção for disparada durante a leitura ou geração de amostras, uma janela será exibida indicando o que pode ter causado o problema.

A geração dos números aleatórios é um elemento de interface composto por outros dois componentes, que são implementados pela classe WidgetRandNumberGenerator e WidgetDistributionParameters.

Figura 18: Tela de obtenção de parâmetros com raiz quadrada como critério de intervalo do histograma

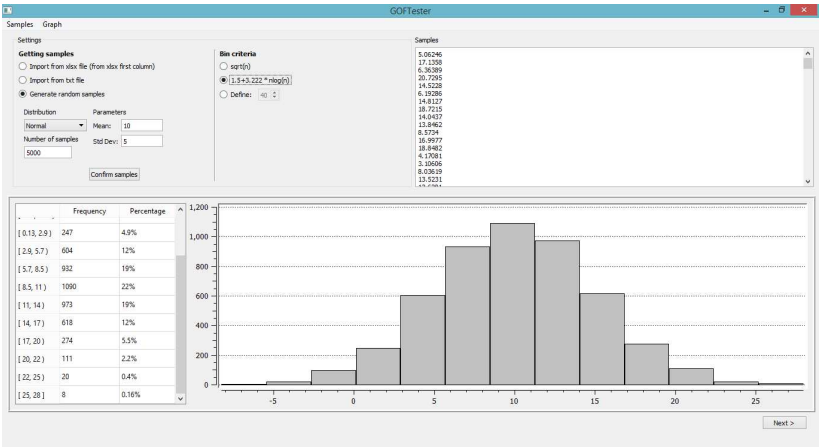


Fonte: Elaborada pelo autor.

Nas Figuras 18 e 19, que são *screenshots* da primeira tela, pode-se notar as amostras geradas a partir de uma distribuição Normal com média 10 e desvio padrão 5. A diferença entre as duas é o critério de definição dos intervalos do histograma. No caso da Figura 18, foi usada a raiz quadrada do número de amostras, o que no caso resultou em 40 intervalos, enquanto que na 19 foi usado o critério chamado de Sturges, o qual resultou em 13 intervalos. Outra opção poderia definir manualmente, sendo que a medida em que um novo valor é inserido, o histograma é automaticamente atualizado.

Após definir as amostras e gerar o histograma, o usuário já pode partir para a próxima tela, onde ele pode escolher quais distribuições serão usadas no teste Qui-Quadrado.

Figura 19: Tela de obtenção de parâmetros com Sturges como critério de intervalo do histograma



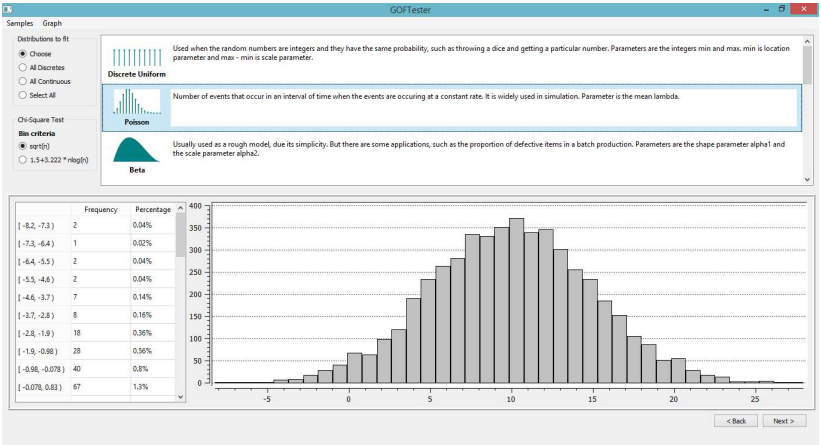
Fonte: Elaborada pelo autor.

4.2.3.7 Classe WidgetDistribToFit

Classe que implementa a interface onde o usuário pode escolher as distribuições que serão aderidas. Informações como aplicações práticas e parâmetros de cada distribuição são exibidas. Existem três formas de se escolher as distribuições: manualmente, selecionar todas discretas ou todas contínuas, selecionas todas distribuições. Além disto, o usuário pode escolher o critério de definição dos intervalos da tabela de frequência do teste de Qui-Quadrado a ser aplicado.

Se o usuário tentar aplicar o teste de aderência em amostras contínuas usando distribuições discretas, uma mensagem de erro irá alertar o usuário que tais distribuições não poderão ser aderidas.

Figura 20: Tela de escolha de distribuição



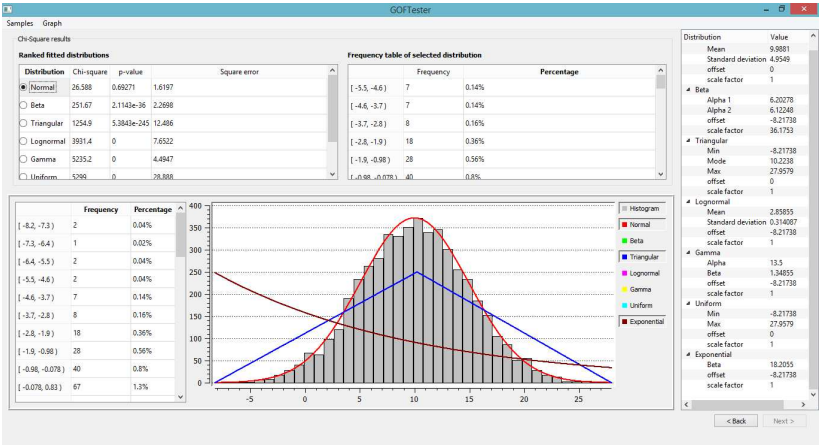
Fonte: Elaborada pelo autor.

4.2.3.8 Classe WidgetChiResults

Classe que exibe todos os resultados do teste de aderência. É apresentada uma tabela com os resultados individuais de cada distribuição, sendo esta tabela ordenada pelo melhor resultado obtido. Uma outra tabela exibe a tabela de frequência do Qui-Quadrado da distribuição selecionada.

No momento em que este componente é apresentado, a janela principal (MainWindow) ativa a opção de plotar as curvas de cada distribuição, e exibe uma tabela com os parâmetros de cada uma delas, além dos fatores de escala e *offset*. Como pode ser visto na Figura 21, mais de uma curva pode ser plotada junto ao histograma.

Figura 21: Tela de resultados do teste de aderência



Fonte: Elaborada pelo autor.

4.2.4 Avaliação

Nesta seção é feita uma breve avaliação dos resultados obtidos pela GOFTester, comparando as estatísticas χ^2 e o $valor_p$ obtidos pelas ferramentas apresentadas no estado da arte. Todas as amostras utilizadas foram geradas pela GOFTester através da funcionalidade de geração de amostras com base em alguma distribuição.

Como pode ser visto nas Tabela 2 e 3, foram aplicados testes para diferentes tipos de distribuições. Ambas tabelas exibem os resultados com base nos três melhores resultados da GOFTester, por isso existe a linha *rank* que indica qual a posição da distribuição na listagem dos melhores resultados ordenados de cada ferramenta.

Pelas tabelas, pode-se notar que a maioria dos resultados da GOFTester se assemelham com os do Input Analyzer, enquanto que os resultados do Crystal Ball e do @Risk obtiveram resultados semelhantes. Um possível fator que pode ter ocasionado nisto é o critério de definição dos intervalos da tabela de frequência do Qui-Quadrado. Em ambos Crystal Ball e @Risk foram usados intervalos de forma que se mantivesse a mesma frequência esperada em cada um deles, enquanto que no GOFTester (e possivelmente no Input Analyzer) este critério não foi utilizado.

Outro fator que pode alterar os resultados é o cálculo dos graus

de liberdade usado pelo Qui-Quadrado. Diferentes formas de se obter os graus de liberdade podem ter sido usadas, o que também pode influenciar nos resultados finais.

Por último, a estimação de parâmetros também pode ter influência nos resultados. Se diferentes técnicas forem utilizadas, evidentemente diferentes resultados serão obtidos.

Embora os resultados foram de forma geral diferentes, o melhor resultado, isto é, a distribuição que melhor aderiu as amostras foi a mesma em todas elas. Como pôde ser notado, não somente a GOFTester obteve resultados diferentes em comparação as outras ferramentas, mas como todas elas tiveram resultados diferentes entre elas. Isto indica que diversos fatores podem afetar nos resultados dos testes e que provavelmente não existe um método ótimo que possa ser aplicado.

Tabela 2: Comparação de resultados de testes Qui-Quadrado usando amostras de distribuição Normal com parâmetros $\mu = 10$ e $\sigma = 5$.

Distribuição Normal				
	GOFTester	Input Analyzer	Crystal Ball	@Risk
rank	1	1	1	1
χ^2	26,588	23,3	36,3040	36,3040
valor_p	0,69271	0,716	0,961	0,9570
Distribuição Beta				
rank	2	2	2	-
χ^2	251,67	63,7	36,6624	-
valor_p	0	$< 0,005$	0,935	-
Distribuição Triangular				
rank	3	6	6	3
χ^2	1.254,9	1.400	1.362,5856	1.277,5552
valor_p	0	$< 0,005$	0	0

Fonte: Elaborada pelo autor.

Tabela 3: Comparação de resultados de testes Qui-Quadrado de distribuição Poisson com parâmetro $\lambda = 2,5$.

Distribuição Poisson				
	GOFTester	Input Analyzer	Crystal Ball	@Risk
rank	1	1	1	1
χ^2	5,222	13,3	5,3722	3,6174
$valor\_p$	0,51567	0,437	0,717	0,7110
Distribuição Exponencial				
rank	2	7	5	3
χ^2	633,44	4.860	46.471,5488	27.818,7552
$valor\_p$	0	$< 0,005$	0	0
Distribuição Beta				
rank	3	3	9	-
χ^2	934,99	95,1	46.505,9328	-
$valor\_p$	0	$< 0,005$	0	-

Fonte: Elaborada pelo autor.

5 CONCLUSÃO

A principal motivação deste trabalho foi a possibilidade de se ter como resultado um software-livre de testes de aderência simples de usar, de modo que ele possa ser usado por estudantes, professores, ou qualquer outra pessoa. A existência da biblioteca libfit foi mais uma motivação para o desenvolvimento desta ferramenta, pois ela poderia se beneficiar das funções estatísticas já existentes na libfit.

Com base nesta ideia, a libfit foi aprimorada com a inclusão de novas distribuições de probabilidade, e a ferramenta de testes de aderência GOFTester, o qual utiliza a libfit, foi desenvolvida neste trabalho. Os requisitos desejáveis na GOFTester foram levantados principalmente com base naquilo que se tem em algumas das principais ferramentas que oferecem testes de aderência, e naquilo que não existe nelas mas se espera ter. Dessa maneira a GOFTester foi desenvolvida visando atender as principais necessidades do usuário alvo.

Como trabalho futuro, algumas funcionalidades ou conceitos podem ser adicionados:

- Adicionar mais distribuições à libfit.
- Adicionar outros testes de aderência à libfit, como o Kolmogorov-Smirnov e o Anderson-Darling
- Exibir outros gráficos no GOFTester, como gráfico de distribuição acumulada.
- Disponibilizar o GOFTester à comunidade *Open-Source*.

REFERÊNCIAS

ANDERSON, T. W.; DARLING, D. A. A test of goodness of fit. **Journal of the American statistical association**, Taylor & Francis, v. 49, n. 268, p. 765–769, 1954.

BANKS, J. et al. **Discrete-event system simulation**. 4. ed. New Jersey: Pearson Prentice Hall, 2005.

BIRTA, L. G.; ARBEZ, G. **Modelling and simulation**: Exploring dynamic system behaviour. London: Springer-Verlag London, 2007.

BRATLEY, P.; FOX, B. L.; SCHRAGE, L. E. **A Guide to Simulation**. New York: Springer-Verlag New York, 1987.

CHENG, R.; CURRIE, C. Resampling methods of analysis in simulation studies. In: WINTER SIMULATION CONFERENCE. **Winter Simulation Conference**. [S.l.], 2009. p. 45–59.

FORMIGHIERI, S. **Uma biblioteca de funções estatísticas para o apoio no desenvolvimento de aplicações com aderência de dados**. 2007. 53 p. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Departamento de Informática e Estatística, Centro Tecnológico, UFSC. Florianópolis. 2007.

FREITAS FILHO, P. J. d. **Introdução à modelagem e simulação de sistemas**: com aplicações em arena. 2. ed. Florianópolis: Visual books, 2008.

KELTON, W David; SADOWSKI, Randall P e SADOWSKI, Deborah A. **Simulation with arena**. New York: McGraw-Hill, 1998.

KOKOSKA, S.; NEVISON, C. **Statistical tables and formulae**. New York: Springer Science & Business Media, 2012.

LAW, A. **Simulation modeling and analysis**. 5. ed. London: McGraw-Hill Education, 2015.

LEEMIS, L. M.; MCQUESTON, J. T. Univariate distribution relationships. **The American Statistician**, Taylor & Francis, v. 62, n. 1, p. 45–53, 2008.

SCOTT, D. W. On optimal and data-based histograms. **Biometrika**, Biometrika Trust, v. 66, n. 3, p. 605–610, 1979.

SHANNON, R. E. Introduction to the art and science of simulation. In: IEEE. **Simulation Conference Proceedings, 1998. Winter**. Washington, 1998. v. 1, p. 7–14.

TENÓRIO, M. B. **Reconhecimento de modelos de probabilidade**. 2005. 77 p. Dissertação (Mestrado) — Departamento de Informática e Estatística, Centro Tecnológico, UFSC. Florianópolis, 2005.

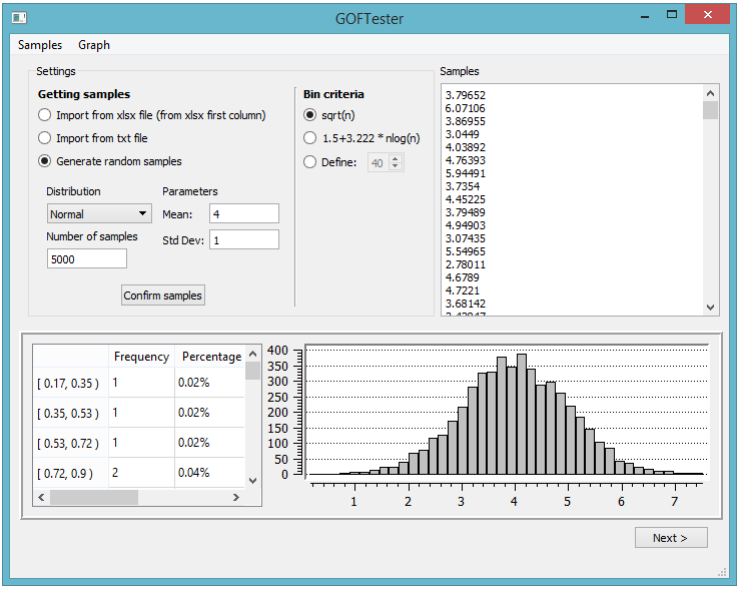
APÊNDICE A – Tutorial de utilização da ferramenta GOFTester

Este tutorial tem como objetivo mostrar um passo-a-passo da utilização da ferramenta de testes de aderência GOFTester.

Passo 1: Obtenção das amostras

A obtenção da amostra pode ser feita através da leitura de arquivo txt ou xlsx, ou gerando amostras aleatórias com base em alguma distribuição. Para uma determinada distribuição escolhida, diferentes parâmetros serão exibidos para terem seus valores inseridos pelo usuário, como pode ser visto na Figura 1.

Figura 1: Tela de obtenção das amostras e geração do histograma



Fonte: Elaborada pelo autor.

Passo 2: Geração do histograma

A geração do histograma é feita pela primeira vez no momento em que se obtém as amostras de dados. Porém, o usuário pode alterar o critério de definição dos intervalos do histograma mesmo após a obtenção das amostras. Na Figura 1 pode-se notar as três opções de definição do critério: raiz quadrada das amostras, critério conhecido como

Sturges, e definição manual. Sempre que o usuário alterar o critério, o histograma é automaticamente atualizado.

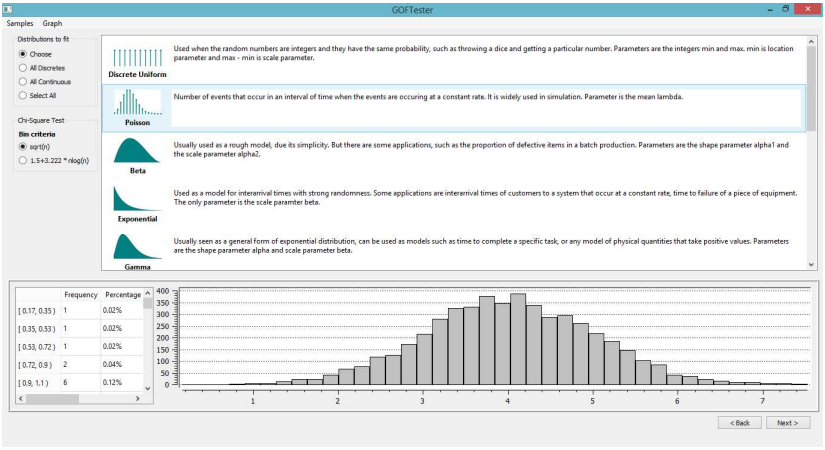
Além do gráfico do histograma, sua tabela de frequências é exibida.

Passo 3: Escolha das distribuições para verificar aderência.

Após obter as amostras e gerar o histograma, o próximo passo é escolher quais distribuições serão usadas para verificar a aderência.

Uma lista de distribuições é exibida (Figura 2) na tela seguinte. O usuário pode tanto escolher as distribuições manualmente, como também optar por selecionar todas elas, todas discretas, ou todas contínuas. Figuras com exemplos das funções de probabilidade de cada distribuições são exibidas, assim como informações sobre aplicações e parâmetros de cada DP. Dessa forma o usuário pode comparar uma distribuição com o gráfico do histograma e ter uma noção prévia de qual distribuição ele deve escolher.

Figura 2: Tela de escolha da distribuição



Fonte: Elaborada pelo autor.

Passo 4: Escolha do critério dos intervalos da tabela de frequências do teste Qui-Quadrado.

Como pode ser visto na Figura 2, nesta tela o usuário pode também escolher qual critério o teste Qui-Quadrado irá utilizar para criar os intervalos da sua tabela de frequências.

Passo 5: Aplicação do teste Qui-Quadrado.

Após escolher as distribuições, ao clicar em *Next* a ferramenta tentará criar todas as distribuições especificadas pelo usuário, e em seguida aplicar o teste de aderência. Se alguma distribuição não puder ser criada, ou não puder ser aplicada no teste, uma janela será exibida indicando quais distribuições falharam.

Passo 6: Entendendo os resultados do teste Qui-Quadrado.

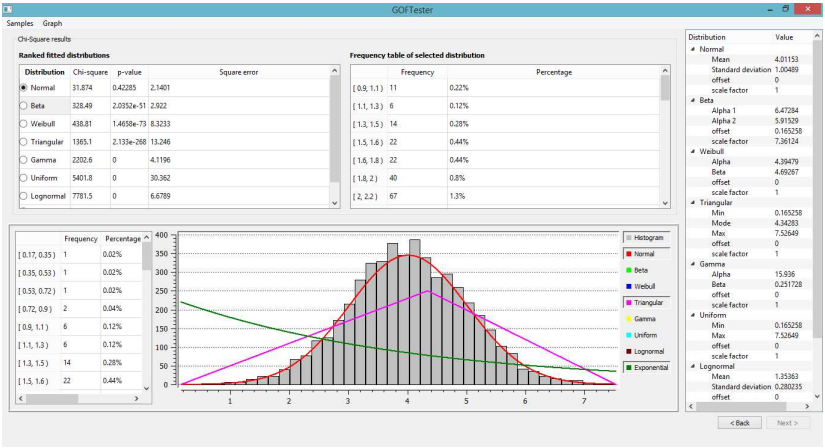
Pela Figura 3, pode-se notar 3 principais componentes: o *Chi-Square results*, uma tabela localizada na parte direta da tela com informações de cada distribuição, e um gráfico na parte inferior da tela.

No *Chi-Square results* encontram-se os resultados do teste Qui-Quadrado. Existe uma tabela de distribuições ordenadas pela melhor aderência, e para cada distribuição selecionada é exibida a sua tabela de frequência utilizada pelo teste.

Na tabela a direita da tela encontram-se informações sobre os parâmetros estimados de cada distribuição. Como cada DP possui um diferente intervalo e escala, para aplicar os testes de aderência é necessário ajustar as amostras para cada DP. Por isso, cada DP tem um valor de *offset* e *scale factor*, que são valores utilizados para colocar as amostras na mesma escala e intervalo das amostras originais.

Por fim, no gráfico do histograma existe uma lista a direita dele com as distribuições ordenadas pelo mesmo critério da tabela de resultados do Qui-Quadrado. O usuário pode selecionar qualquer uma destas distribuições para que ela seja plotada no gráfico. Desta forma é possível comparar visualmente a curva da DP com o histograma.

Figura 3: Tela de resultados do teste de aderência



Fonte: Elaborada pelo autor.

Passo 7: Exportando dados.

A partir do momento em que as amostras são geradas, o usuário pode em qualquer momento salvar as amostras em um arquivo txt ou xlsx. Para isto, basta clicar no menu *Samples* e em *Export*.

Além das amostras, o gráfico também pode ser salvo, sendo que o usuário pode escolher entre os formatos png, jpg, svg, ou pdf. O gráfico no estado atual, isto é, com determinadas curvas plotadas, será salvo no arquivo. Para salvar, basta clicar no menu *Graph* e em *Export graph*.

APÊNDICE B – Artigo

GOFTester: Aprimoramento e Implementação da Biblioteca Libfit em uma Ferramenta para Testes de Aderência

Diego A. de Oliveira

Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 476 – 88.040-900 – Florianópolis – SC – Brasil

diegohkd@hotmail.com

Abstract. *Goodness-of-fit tests are tools frequently employed with the purpose of verifying whether a given sample of random data fits some theoretical probability distribution. This research focuses mostly in improving the goodness-of-fit library called libfit, by adding Qui-Square tests for some discrete distributions, and designing and implementing a goodness-of-fit tool using C++ and Qt framework. This tool, called GOFTester, uses libfit in order to perform goodness-of-fit tests and provide satisfactory results to the user. paper.*

Resumo. *Os testes de aderência são procedimentos estatísticos frequentemente empregados quando se deseja verificar se uma dada amostra aleatória de dados adere alguma distribuição teórica. Este trabalho concentrou-se principalmente no aperfeiçoamento da biblioteca de testes de aderência libfit, através da inclusão do teste Qui-Quadrado para algumas distribuições de probabilidade discretas, e no projeto e desenvolvimento de uma ferramenta em C++ utilizando o framework Qt. Esta ferramenta, chamada GOFTester, faz uso da libfit para realizar os testes de aderência e gerar resultados satisfatórios para o usuário.*

1. Introdução

A sociedade atual, em contraste com civilizações mais antigas, já chegou a um patamar em que muitas conquistas foram atingidas e grandes feitos já foram realizados, como por exemplo a invenção dos aviões, que hoje carregam centenas de pessoas. Porém, junto a essa evolução, desafios maiores e mais complexos vão surgindo, como no caso dos aviões, no qual um grande desafio era colocá-los no ar com algum piloto que já possuísse alguma experiência de voo sem nunca ter pilotado antes. Quando alguém se depara com problemas desse tipo e tenta contorná-los, diversos fatores podem dificultá-los e torná-los inviáveis ou até mesmo impossíveis de serem tratados diretamente.

Para auxiliar nesta tarefa, uma opção é simular os sistemas em questão. Diversas variáveis podem estar envolvidas nesses sistemas, sendo que para representá-las de forma fiel pode-se, por exemplo, obter amostras e modelos estatísticos que melhor represente-as, como as distribuições de probabilidade. Para obter essas distribuições, não basta apenas estimar os seus parâmetros com base nas amostras, mas também avaliar o quão bem elas representam as variáveis com aqueles parâmetros estimados, isto é, se as

amostras aderem ou não as distribuições teóricas de probabilidade. Um dos métodos de se realizar essa verificação é através de testes de aderência. Os testes podem ser realizados computacionalmente, como é feito pela biblioteca desenvolvida em C++, libfit.

1.1. Justificativa

Os testes de aderência são procedimentos estatísticos que já foram bastante estudados na literatura. Além de existir uma infinidade de estudos realizados, também existem muitas ferramentas desenvolvidas que proveem a realização desses testes. Algumas dessas ferramentas são acessíveis aos usuários comuns, como os estudantes, enquanto que outras já não são tão acessíveis. Além do fato delas muitas vezes terem custos elevados, geralmente são de propósito mais geral, como no caso de alguns softwares para se realizar modelagem e simulação de sistemas.

Se um usuário deseja realizar somente testes de aderência, ele acaba se encontrando em uma situação no qual é preciso obter softwares mais robustos que possuem testes estatísticos acoplados ao seu sistema. A opção de se obter uma ferramenta gratuita que realiza os testes, no qual o usuário pode comparar resultados através de dados estatísticos ou de forma visual com gráficos, mostra-se uma boa solução para esse problema.

Essa ferramenta, cujo nome atribuído é GOFTester, foi projetada e desenvolvida de forma a usar a biblioteca libfit para realizar os testes de aderência. A biblioteca, graças a sua modularidade, pode ser acoplada ao sistema de forma que se possa reaproveitar todas suas funções implementadas, sem a necessidade de reimplementar funções de testes de aderência. Assim, pode-se notar que essa portabilidade permite que qualquer desenvolvedor reutilize o código implementado na libfit em algum programa que ele esteja desenvolvendo, sem a necessidade de que o programa fique dependente de outras ferramentas.

2. Objetivos

O objetivo deste trabalho é aprimorar a biblioteca libfit e implementar em uma ferramenta para testes de aderência. Para isto, os seguintes objetivos específicos foram atingidos:

- Identificar quais distribuições não estão presentes na libfit para realizar o teste Qui-Quadrado, e definir quais serão implementadas.
- Adicionar à libfit as distribuições selecionadas para serem implementadas.
- Projetar e implementar uma ferramenta que faça uso da libfit para realizar testes de aderência.
- Avaliar esta ferramenta.

3. Fundamentação

Para aprimorar a biblioteca libfit e implementá-la na GOFTester, três assuntos merecem destaque: as distribuições de probabilidades, estimativa de parâmetros por Máxima Verossimilhança e os testes de aderência.

3.1. Distribuições de Probabilidade

Existem inúmeras classes de distribuições de probabilidades, cada uma apresentando uma característica que se mostra útil para uma diferente aplicação. Certas distribuições possuem propriedades que têm se mostrado bastante úteis em simulações de sistemas, sendo assim mais relevantes para este trabalho, sendo que elas são separadas entre as discretas e as contínuas.

Distribuições de probabilidade discretas são aquelas cujas variáveis podem ter um número finito de valores. Tais distribuições normalmente representam fenômenos como o número de ocorrências de algum evento, por exemplo o lançamento de dados, ou a quantidade de eventos que ocorrem em um intervalo de tempo. Existem diversas distribuições discretas, com diferentes aplicações, como as distribuições Binomial, Geométrica, Bernoulli, etc. Neste trabalho serão abordadas as distribuições Uniforme Discreta e a Poisson, sendo ambas bastante usadas em simulações de sistemas, principalmente a Poisson, utilizada por exemplo na Teoria de Filas [Freitas Filho 2008].

As distribuições contínuas são aquelas cujas variáveis podem ter infinitos valores dentro de algum intervalo bem definido. Existem diversas utilidades para cada uma delas, como no caso da Exponencial que geralmente é usada para representar o tempo entre dois eventos que apresentem uma forte aleatoriedade, e a Weibull, que é frequentemente usada por engenheiros para representar uma variável aleatória de algum sistema ou equipamento.

3.2. Estimação de Máxima Verossimilhança

A Estimação de Máxima Verossimilhança, ou *Maximum Likelihood Estimation* (MLE), é uma técnica usada para obter, isto é, estimar os parâmetros de algum modelo estatístico. Como mencionado por Law (2015), esse procedimento usufrui de algumas ferramentas para a estimação dos parâmetros de algumas distribuições de probabilidade, como a função de log-verossimilhança ou através de diferenciação. Porém, essas duas técnicas não são aplicáveis para todas as distribuições, sendo que em alguns casos é preciso recorrer a métodos numéricos. Na literatura já existem fórmulas e métodos para se obter os parâmetros de todas as distribuições usadas neste trabalho, como apresentado na Tabela 1.

Finalmente, feita a estimação dos parâmetros, o próximo passo é a realização dos testes de aderência para verificar se a amostra segue alguma distribuição de probabilidade.

3.3. Testes de Aderência

Para se medir o quão bem uma amostra segue um determinado modelo ou distribuição, uma alternativa são os testes de aderência, que são uma classe de testes de hipóteses. De modo geral, o objetivo de um teste de hipóteses é verificar se alguma afirmação em relação a um conjunto de dados é verdadeira. Essa afirmação é chamada de hipótese nula, denotada por H_0, enquanto que a hipótese que rejeita, isto é, que contraria H_0, é denotada por H_1. Ao declarar uma afirmação, seja ela H_0 ou H_1, a ideia é criar um modelo de decisão sobre um conjunto de dados para concluir se essa afirmação é realmente verdadeira. Se for concluído que a afirmação estava errada, o que se diz é que foi um erro dizer que ela era verdadeira.

Tabela 1 - Estimadores dos Parâmetros de Distribuições de Probabilidade Discretas e Contínuas.

Distribuição	Parâmetro(s)	Estimador
Uniforme Discreta	i, j	$i = \min_{1 \le k \le n} X_k, \quad j = \max_{1 \le k \le n} X_k$
Poisson	λ	$\lambda = \bar{X}(n)$
Uniforme	a, b	$a = \min_{1 \le i \le n} X_i, \quad j = \max_{1 \le i \le n} X_i$
Exponencial	β	$\beta = \bar{X}(n)$
Gamma	α, β	$\alpha = \frac{\bar{X}^2}{S^2}, \quad \beta = \frac{S^2}{\bar{X}}$
Weibull	α, β	α: obtido utilizando o método de Newton $\beta = \left(\frac{\sum_{i=1}^{n} X_i^{\alpha}}{n} \right)^{1/\alpha}$
Normal	μ, σ	$\mu = \bar{X}(n) \quad \sigma = \left[\frac{n-1}{n} S^2(n) \right]^{\frac{1}{2}}$
Lognormal	μ, σ	ver seção 2.1.2.6
Beta	α, β	$\alpha = \bar{X} \left[\left[\frac{\bar{X}(1-\bar{X})}{S^2} \right] - 1 \right]$ $\beta = (\bar{X} - 1) \left[\left[\frac{\bar{X}(1-\bar{X})}{S^2} \right] - 1 \right]$
Triangular	a, b	$a = \min_{1 \le i \le n} X_i, \quad b = \max_{1 \le i \le n} X_i$

Os testes de aderência são testes de hipóteses em que se deseja verificar se um conjunto de dados segue (adere) alguma distribuição de probabilidade [Law 2015]. Por se tratar de um tipo de teste de hipóteses, visa-se verificar se a hipótese nula H_0 é verdadeira. Nesse caso, a hipótese nula é a seguinte:

H_0: a amostra é composta por variáveis aleatórias com função distribuição acumulada F.

Existem diversos tipos de teste de aderência, como o Qui-Quadrado, Kolmogorov-Smirnov e o Anderson-Darling. Neste trabalho o enfoque é no Qui-Quadrado.

3.4. Teste Qui-Quadrado

O teste Qui-Quadrado tem como primeiro passo a separação das amostras em k classes (ou intervalos), no qual para cada classe j existe uma probabilidade teórica p_j associada, isto é, um valor que representa a proporção de dados naquele intervalo referente a distribuição que se está verificando a aderência.

Por conseguinte, para se obter a estatística do teste Qui-Quadrado χ^2, calcula-se a equação [Freitas Filho 2008]:

$$\chi^2 = \sum_{j=1}^{k} \frac{(fo_j - fe_j)^2}{fe_j}$$

Onde fo_j é a frequência observada na classe j, e fe_j é a frequência esperada $fe_j = np_j$, sendo n o total de amostras.

Quanto maior χ^2, pode-se se dizer que menos a amostra adere a distribuição em questão. Portanto, se $\chi^2 = 0$, então a amostra segue perfeitamente a distribuição de probabilidade.

Entretanto, como geralmente uma amostra não segue uma distribuição com $\chi^2 = 0$, o que se verifica é se a amostra segue aproximadamente a distribuição com $v = k - 1 - p$ graus de liberdade, sendo p a quantidade de parâmetros da distribuição teórica com adição de mais uma unidade.

Uma das formas de realizar essa verificação é através da obtenção do $valor\_p$, que em termos gerais indica a probabilidade em se obter com a distribuição teórica um mesmo valor obtido com a amostra, sendo que quanto menor o p, menor é essa probabilidade. Assim, define-se um nível de significância α, que é um indicador da probabilidade de se rejeitar H_0 sendo ela verdadeira, ou seja, em se cometer um erro do tipo I, e verifica-se se $valor\_p \leq \alpha$. Se verdadeira, então a hipótese nula é rejeitada.

O Qui-Quadrado é o único teste de aderência implementado na libfit. Mais detalhes sobre o teste e sua implementação são apresentados em Formighieri (2007).

4. Proposta

Neste capítulo serão apresentados os problemas e as modelagens das soluções para tanto o aprimoramento da libfit quanto para a implementação da ferramenta de testes de aderência.

4.1. Biblioteca Libfit

Por se tratar de uma biblioteca, isto é, um conjunto de funções aplicáveis em outros programas, é desejável que ela possua uma interface simples e concisa, de forma que garanta uma intercomunicação eficiente e segura entre a aplicação e a biblioteca. Nessa interface, alguns detalhes internos não precisam (muitas vezes não devem) ser acessíveis pela aplicação, induzindo a uma abstração daquilo que é irrelevante para o programa.

Como a realização do teste de aderência é o principal objetivo da biblioteca, aquilo que deve ser acessível à aplicação deve não somente permitir aplicar o teste, como também fornecer dados estatísticos usados durante ou como resultado final do processo de aderência. Na Figura 1 pode-se ver as etapas mais relevantes de aplicação do teste de aderência fornecido pela libfit. Com base no fluxograma, é possível notar três etapas: obtenção da distribuição de probabilidade, geração da tabela de frequências e aplicação do teste de aderência.

4.2. Melhorias da Libfit

Como mencionado anteriormente, a libfit só possui distribuições contínuas, sendo que todas as funções implementadas assumem dados contínuos, isto é, que pertencem ao conjunto dos números reais. A proposta de melhoria da biblioteca é em acrescentar os testes de aderência para algumas distribuições discretas, sendo que as distribuições implementadas foram a Poisson e a Uniforme Discreta.

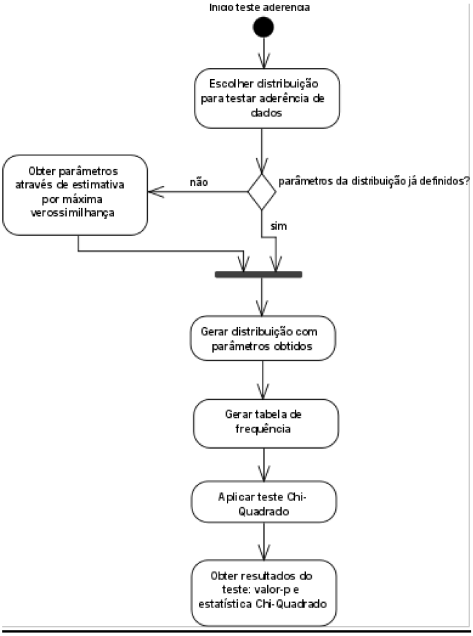


Figura 1 - Diagrama de atividades para teste de aderência versão original libfit

A melhoria implicou na implementação de funções como as seguintes:

- **cumulative_function**: Função acumulada da distribuição. É usada no teste de aderência no cálculo de frequência esperada.
- **probability_function**: Função de probabilidade, ou função massa de probabilidade, no caso das distribuições discretas. Pode ser utilizada para plotar curva da distribuição.
- **get_random**: Gera número aleatório de acordo com a distribuição.
- **set_params_mle**: Estima os parâmetros da distribuição usando MLE.

4.3. Desenvolvimento da Ferramenta GOFTester

Para tratar do desenvolvimento da GOFTester, foram usados técnicas de engenharia de software para realizar o projeto e implementação da ferramenta, como a análise de requisitos, casos de uso e diagrama de classe do projeto.

O diagrama de classes do projeto pode ser visualizado na Figura 2**.** Apesar de ser um diagrama de classes de projeto, foram omitidas informações como atributos e funções, para efeito de simplificação e melhor entendimento das relações entre as classes. A classe MainWindow representa a janela principal do programa, todas as etapas de aplicação do

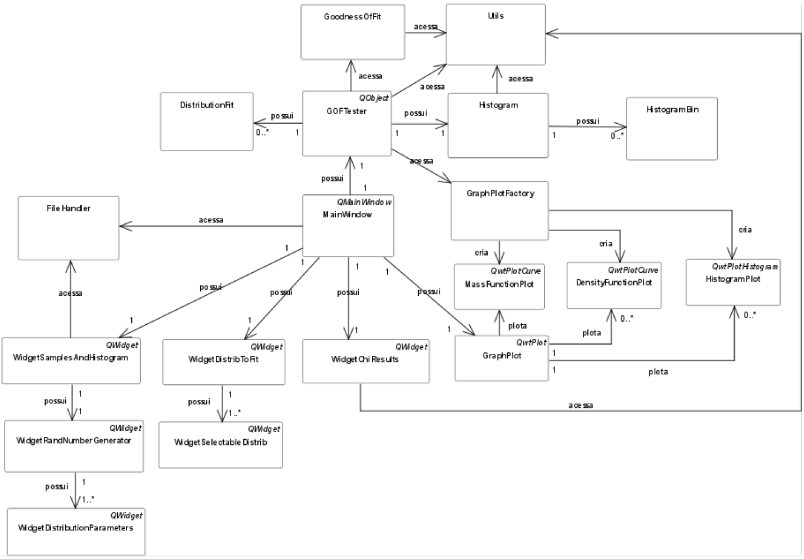


Figura 2 - Diagrama de classe da ferramenta GOFTester

teste de aderência, desde a obtenção das amostras até a exibição dos resultados dos testes, serão feitas nessa janela. Porém, cada funcionalidade, como a geração de amostras aleatórias, será atribuída a componentes de interface separados, conhecidos como Widgets.

Cada um desses componentes é uma classe que herda a classe QWidget, que é nativa do Qt. Dessa forma evita-se uma grande sobrecarga de funcionalidades que poderiam ser todas implementadas na MainWindow. Essas classes, além daquelas que herdam classes de plotagem da biblioteca Qwt, formam a visão da arquitetura do programa. Como mencionado anteriormente, a visão exerce também o papel de controlador, sendo que a MainWindow ficará responsável pela interface entre o modelo e todos os componentes da visão.

Exercendo um papel importante no programa, fica atribuído a classe GOFTester a responsabilidade de gerenciar as principais funcionalidades, como obter o histograma das amostras, usar as amostras para gerar distribuições de probabilidade, realizar requisições de aplicação de teste de aderência nas distribuições e amostras, etc. Ela herda QObject para que possa se comunicar com a MainWindow através de *signals* e *slots*.

Outras classes importantes são: GoodnessOfFit, que é responsável por acessar a biblioteca libift para realizar testes de aderência; DistributionFit, que representa uma distribuição de probabilidade que, além das distribuições geradas pela libfit a partir das amostras, contém também os resultados do teste de aderência, amostras, fatores de escala e offset das amostras, etc; e Histogram, classe que contém instâncias da classe HistogramBin, que são os intervalos do histograma.

APÊNDICE C – Código-Fonte

Listagem C.1: base_distrib.cpp

```
//
    ↪ ------------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪

#include <distribution/base_distrib.h>
#include <cstdlib>
#include <ctime>
#include <algorithm>

using namespace fit;

/*
base_distrib::base_distrib(const data_set& values)
{
    _parameters = set_params_mle(values);
    std::srand(std::time(0));
}
*/
```

```
base_distrib::base_distrib()
{
    std::srand(std::time(0));
}

base_distrib::base_distrib(const params_list& params)
    : _parameters(params)
{
    std::srand(std::time(0));
}

bool base_distrib::check_values(const data_set& values)
    ↪ const
{
    for (data_set::const_iterator it = values.begin(); it !=
        ↪  values.end(); it++)
        if (!in_range(*it))
            return false;

    return true;
}

data_type base_distrib::expected_freq(data_type from_number,
    ↪  data_type to_number, unsigned int total) const
{
    return total * (cumulative_function(to_number) -
        ↪ cumulative_function(from_number));
}

data_type base_distrib::random_number() const
{
    data_type ran = std::rand() / ( RAND_MAX + 1.0 );
    if (ran == 0.0)
        ran = 0.000000000001;
    else if (ran == 1.0)
        ran = 0.999999999999;
    return ran;
}
```

Listagem C.2: base_distrib.h

```
//
    ↪ ------------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
```

```
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪

#ifndef BASE_DISTRIB_H_
#define BASE_DISTRIB_H_

#include "../fit_defs.h"
#include <vector>

namespace fit
{
typedef std::vector<data_type> params_list;
//the base probability distribution
class base_distrib
{
protected:
    params_list _parameters;
    sample_type _samples_type;

    /* Uniform(0;1) number generator */
    data_type random_number() const;

    /* Verifies weather value is in distribution possible
        ↪ range.
     * In this case range is (-infinite, +infinite) */
    virtual inline bool in_range(data_type value) const { (
        ↪ void)value; return true; }

    base_distrib();
    base_distrib(const params_list& params); //gets
        ↪ distribution parameters from params.
    //base_distrib(const data_set& values); //gets
```

```
        ↪ distribution parameters from MLE

public:

    //Return a clone of de distribuition allocated on heap
    virtual base_distrib *clone() = 0;

    virtual std::string name() const = 0;
    virtual std::vector<std::string> paramNames() const = 0;

    virtual ~base_distrib() {}

    virtual data_type get_mean() const = 0;
    virtual data_type get_variance() const = 0;
    virtual data_type get_mode() const
    {
        throw distribution_exception("Mode does not uniquely
            ↪ exist.");
    }

    /* expected_freq = total * (cumulative_function(
        ↪ to_number) - cumulative_function(from_number)) */
    data_type expected_freq(data_type from_number, data_type
        ↪ to_number, unsigned int total = 1) const;

    virtual data_type cumulative_function(data_type number)
        ↪ const = 0; //cumulative distribution function
    virtual data_type probability_function(data_type number)
        ↪ const = 0; //probability density/mass function

    /* Sets distribution parameters through maximum
        ↪ likelihood procedures */
    virtual void set_params_mle(const data_set& values) = 0;

    /* Gets a random value according to distribution */
    virtual data_type get_random() const
    {
        throw distribution_exception("No random number
            ↪ generator defined for this function.");
    }

    virtual bool check_parameters(const params_list& params)
        ↪ const = 0;
    bool check_values(const data_set& values) const;

    unsigned int get_params_number() const { return
        ↪ _parameters.size(); }
    params_list get_params() const { return _parameters; }

    sample_type get_samples_type() const { return
        ↪ _samples_type; }
};
```

```
}

#endif /*BASE_DISTRIB_H_*/
```

Listagem C.3: beta_distrib.cpp

```
//
    ↪ ------------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪

#include <distribution/beta_distrib.h>
#include <common/fit_utils.hpp>

using namespace fit;

beta_distrib::beta_distrib(const beta_distrib& other) :
gamm1(new gamm_distrib(*other.gamm1)),
    gamm2(new gamm_distrib(*other.gamm2)),
    base_distrib(other._parameters)
{
}
```

```
beta_distrib::beta_distrib(const data_set& values)
{
    _samples_type = sample_type::CONTINUOUS;

    _parameters.resize(PARAMS_NUMBER);
    set_params_mle(values);

    params_list par;
    par.push_back(_parameters[ALPHA1]);
    par.push_back(1.0);
    gamm1 = new gamm_distrib(par);

    par[0] = _parameters[ALPHA2];
    gamm2 = new gamm_distrib(par);
}

beta_distrib::beta_distrib(const params_list& params)
{
    _samples_type = sample_type::CONTINUOUS;

    if (!check_parameters(params))
        throw distribution_exception("Invalid parameters on
            ↪ Beta distribution.");

    _parameters = params;

    params_list par;
    par.push_back(_parameters[ALPHA1]);
    par.push_back(1.0);
    gamm1 = new gamm_distrib(par);

    par[0] = _parameters[ALPHA2];
    gamm2 = new gamm_distrib(par);
}

base_distrib *beta_distrib::clone()
{
    return new beta_distrib(*this);
}

std::string beta_distrib::name() const
{
    return "Beta";
}

std::vector<std::string> beta_distrib::paramNames() const
{
    std::vector<std::string> paramNames = std::vector<std::
        ↪ string>();
    paramNames.push_back("Alpha 1");
    paramNames.push_back("Alpha 2");
```

```
    return paramNames;
}

beta_distrib::~beta_distrib()
{
    delete gamm1;
    delete gamm2;
}

data_type beta_distrib::get_mean() const
{
    data_type alpha = _parameters[ALPHA1];
    return alpha / (alpha + _parameters[ALPHA2]);
}

data_type beta_distrib::get_variance() const
{
    data_type alpha1 = _parameters[ALPHA1];
    data_type alpha2 = _parameters[ALPHA2];
    data_type den_a = (alpha1 + alpha2) * (alpha1 + alpha2);
    return (alpha1 * alpha2) / ( den_a * (alpha1 + alpha2 +
        ↪ 1) );
}

data_type beta_distrib::get_mode() const
{
    data_type alpha1 = _parameters[ALPHA1];
    data_type alpha2 = _parameters[ALPHA2];

    if ((alpha1 == alpha2) && (alpha1 == 1))
        throw distribution_exception("Mode does not uniquely
            ↪  exist.");

    if ((alpha1 < 1) && (alpha2 < 1))
        return 0; // or return 1;

    if (((alpha1 < 1) && (alpha2 >= 1)) || ((alpha1 == 1) &&
        ↪  (alpha2 > 1)))
        return 0;

    if (((alpha1 >= 1) && (alpha2 < 1)) || ((alpha1 > 1) &&
        ↪ (alpha2 == 1)))
        return 1;

    if ((alpha1 > 1) && (alpha2 > 1))
        return (alpha1 - 1) / (alpha1 + alpha2 - 2);

    return 0;
}

data_type beta_distrib::cumulative_function(data_type number
```

```
    ↪ ) const
{
    return utils::betai(_parameters[ALPHA1], _parameters[
        ↪ ALPHA2], number);
}

data_type beta_distrib::beta(data_type z, data_type w)
{
    return std::exp(utils::gammln(z) + utils::gammln(w) -
        ↪ utils::gammln(z+w));
}

data_type beta_distrib::probability_function(data_type
    ↪ number) const
{
    if ( (number <= 0) || (number >= 1) )
        return 0;

    data_type alpha1 = _parameters[ALPHA1];
    data_type alpha2 = _parameters[ALPHA2];
    data_type fracnum = std::pow(number, alpha1 - 1) * std::
        ↪ pow(1 - number, alpha2 - 1);
    return fracnum / beta_distrib::beta(alpha1, alpha2);
}

void beta_distrib::set_params_mle(const data_set& values)
{
    if (!check_values(values))
        throw distribution_exception("Invalid value on Beta
            ↪ distribution.");

    data_type media = utils::_mean(values);
    data_type varian = utils::_variance(media, values);
    _parameters[ALPHA1] = media * (((media * (1 - media))/
        ↪ varian) - 1);
    _parameters[ALPHA2] = (1 - media) * (((media * (1 -
        ↪ media))/varian) - 1);

    if (!check_parameters(_parameters))
        throw distribution_exception("Invalid parameters on
            ↪ Beta distribution.");
}

data_type beta_distrib::get_random() const
{
    data_type Y1 = gamm1->get_random();
    data_type Y2 = gamm2->get_random();
    return Y1 / (Y1 + Y2);
}

bool beta_distrib::check_parameters(const params_list&
    ↪ params) const
```

```
{
    return ((params.size() == PARAMS_NUMBER) &&
            (params.at(ALPHA1) > 0) &&
            (params.at(ALPHA2) > 0));
}
```

Listagem C.4: beta_distrib.h

```
//
    ↪ ------------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪

#ifndef BETA_DISTRIB_H_
#define BETA_DISTRIB_H_

/*
 * Beta Distribution
 */

#include "gamm_distrib.h"

namespace fit
```

```
{
class beta_distrib : public base_distrib
{
private:
    gamm_distrib *gamm1, *gamm2;

protected:
    inline bool in_range(data_type value) const { return (
        ↪ value >= 0) && (value <= 1); }
    static data_type beta(data_type z, data_type w);

public:
    enum _beta_params
    {
        ALPHA1,
        ALPHA2,
        PARAMS_NUMBER
    };

    beta_distrib(const beta_distrib& other);

    beta_distrib(const data_set& values); //gets
        ↪ distribution parameters from MLE
    beta_distrib(const params_list& params); //gets
        ↪ distribution parameters from params.

//  Return a clone of de distribuition allocated on heap
    base_distrib *clone();

    std::string name() const;
    std::vector<std::string> paramNames() const;

    ~beta_distrib();

    data_type get_mean() const;
    data_type get_variance() const;
    data_type get_mode() const;

    data_type cumulative_function(data_type number) const;
        ↪ //cumulative distribution function
    data_type probability_function(data_type number) const;
        ↪ //probability density function

    /* Sets distribution parameters through maximum
        ↪ likelihood procedures */
    void set_params_mle(const data_set& values);

    /* Gets a random value according to distribution */
    data_type get_random() const;

    bool check_parameters(const params_list& params) const;
};
```

```
}

#endif /*BETA_DISTRIB_H_*/
```

Listagem C.5: defs.h

```
#ifndef DEFS_H
#define DEFS_H

#include <QString>

/* Exceptions */
struct file_exception
{
    QString msg;
    file_exception(QString _msg) : msg(_msg) {}
};

#endif // DEFS_H
```

Listagem C.6: density_function_plot.cpp

```
#include "view/plot/density_function_plot.h"

DensityFunctionPlot::DensityFunctionPlot(const QString title
    ↪ , const QColor& color)
{
    this->setTitle(title);
    this->setPen( color, 2 ),
    this->setRenderHint( QwtPlotItem::RenderAntialiased,
        ↪ true );
}

void DensityFunctionPlot::setValues(QList<QPointF> points)
{
    QPolygonF points_plot;
    foreach(QPointF point, points)
    {
        points_plot << point;
    }
    setSamples(points_plot);
}
```

Listagem C.7: density_function_plot.h

```
#ifndef DENSITYFUNCTIONPLOT_H
#define DENSITYFUNCTIONPLOT_H

#include "qwt_plot_curve.h"

class DensityFunctionPlot: public QwtPlotCurve
```

```
{
public:
    DensityFunctionPlot(const QString title, const QColor&
        ↪ color);
    void setValues(QList<QPointF> points);
};

#endif // DENSITYFUNCTIONPLOT_H
```

Listagem C.8: discrete_uniform_distrib.cpp

```
#include "distribution/discrete_uniform_distrib.h"

#include "common/fit_utils.hpp"

#include <cmath>
#include <algorithm>

using namespace fit;

discrete_uniform_distrib::discrete_uniform_distrib(const
    ↪ data_set& values)
{
    if(utils::_check_samples_type(values) != sample_type::
        ↪ DISCRETE)
        throw distribution_exception("Discrete Uniform
            ↪ samples must be integer.");
    _samples_type = sample_type::DISCRETE;
    _parameters.resize(PARAMS_NUMBER);
    set_params_mle(values);
}

base_distrib *discrete_uniform_distrib::clone()
{
    return new discrete_uniform_distrib(*this);
}

std::string discrete_uniform_distrib::name() const
{
    return "Discrete Uniform";
}

std::vector<std::string> discrete_uniform_distrib::
    ↪ paramNames() const
{
    std::vector<std::string> paramNames = std::vector<std::
        ↪ string>();
    paramNames.push_back("Min");
    paramNames.push_back("Max");
    return paramNames;
}
```

```
discrete_uniform_distrib::discrete_uniform_distrib(const
    ↪ params_list& params)
{
    _samples_type = sample_type::DISCRETE;
    if (!check_parameters(params))
        throw distribution_exception("Invalid parameters on
            ↪ discrete uniform distribution.");

    _parameters = params;
}

data_type discrete_uniform_distrib::cumulative_function(
    ↪ data_type number) const
{
    data_type min = _parameters[MIN];
    data_type max = _parameters[MAX];

    if(number < min)
        return 0;
    if(min <= number && number <= max)
        return (floor(number)- min + 1)/(max - min + 1);
    if(max < number)
        return 1;
    return 1;
}

data_type discrete_uniform_distrib::probability_function(
    ↪ data_type number) const
{
    data_type min = _parameters[MIN];
    data_type max = _parameters[MAX];

    if(min <= number && number <= max)
        return 1/(max - min + 1);

    return 0;
}

void discrete_uniform_distrib::set_params_mle(const data_set
    ↪ & values)
{
    _parameters[MIN] = *min_element(values.begin(), values.
        ↪ end());
    _parameters[MAX] = *max_element(values.begin(), values.
        ↪ end());
}

data_type discrete_uniform_distrib::get_mean() const
{
    return (_parameters[MIN] + _parameters[MAX])/2;
}
```

```
data_type discrete_uniform_distrib::get_variance() const
{
    return pow(_parameters[MAX] - _parameters[MIN] + 1, 2)
        ↪ /(12);
}

data_type discrete_uniform_distrib::get_mode() const
{
    throw distribution_exception("Mode does not uniquely
        ↪ exist.");
}


data_type discrete_uniform_distrib::get_random() const
{
    data_type min = _parameters[MIN];
    data_type max = _parameters[MAX];
    data_type u = random_number();

    return min + floor((max - min + 1)*u);
}

bool discrete_uniform_distrib::check_parameters(const
    ↪ params_list& params) const
{
    return params[MIN] <= params[MAX];
}
```

Listagem C.9: discrete_uniform_distrib.h

```
#ifndef DISCRETE_UNIFORM_DISTRIB_H
#define DISCRETE_UNIFORM_DISTRIB_H

#endif // DISCRETE_UNIFORM_DISTRIB_H

/*
 * Discrete Uniform Distribution
 */

#include "base_distrib.h"

namespace fit
{
class discrete_uniform_distrib: public base_distrib
{
public:
    enum _discrete_uniform_params
    {
        MIN,
        MAX,
        PARAMS_NUMBER
    };
```

```
    discrete_uniform_distrib(const data_set& values); //gets
        ↪ distribution parameters from MLE
    discrete_uniform_distrib(const params_list& params); //
        ↪ gets distribution parameters from params.

    //  Return a clone of de distribuition allocated on heap
    base_distrib *clone();

    std::string name() const;
    std::vector<std::string> paramNames() const;

    virtual ~discrete_uniform_distrib() {}

    virtual data_type get_mean() const;
    virtual data_type get_variance() const;
    virtual data_type get_mode() const;

    virtual data_type cumulative_function(data_type number)
        ↪ const;
    virtual data_type probability_function(data_type number)
        ↪ const; //probability mass function

    /* Sets distribution parameters through maximum
        ↪ likelihood procedures */
    void set_params_mle(const data_set& values);

    /* Gets a random value according to distribution */
    virtual data_type get_random() const;

    bool check_parameters(const params_list& params) const;
};
}
```

Listagem C.10: distribution_fit.cpp

```
#include "model/distribution_fit.h"

#include <list>
#include <vector>

#include "utils/utils.h"

DistributionFit::DistributionFit(fit::distrib_ptr
    ↪ distribution, fit::_distribution_type type,
                                 double offset, double
                                     ↪ scale_factor, QList<
                                     ↪ double> samples):
   type_(type), offset_(offset), scale_factor_(scale_factor)
       ↪ ,
    samples_(samples)
{
    // init class members
```

```
    distribution_ = QSharedPointer<fit::base_distrib>(
        ↪ distribution.release());
    min_ = samples_.first();
    max_ = samples.last();

    // get and set paramaters' names and values
    QVector<std::string> temp_params_names = QVector<std::
        ↪ string>::fromStdVector(distribution_->paramNames
        ↪ ());
    QVector<double> temp_params_values = QVector<double>::
        ↪ fromStdVector(distribution_->get_params());
    for(int i = 0; i < temp_params_names.size(); i++)
    {
        params_.push_back(QPair<QString, double>(QString::
            ↪ fromStdString(temp_params_names.at(i)),
            ↪ temp_params_values.at(i)));
    }
}

// get scaled and offsetted y-axis points of mass/density
    ↪ function
QList<QPointF> DistributionFit::getProbabilityFunctionPoints
    ↪ (int n_points, double y_mean_hist, double width_bin,
    ↪ double n_samples)
{
    // set y scale factor -> distribution dependent
    double y_mean_distrib = distribution_->
        ↪ probability_function(distribution_->get_mean());
    double scale_y = 1;
    if(type_ != fit::UNIFORM && type_ != fit::
        ↪ DISCRETE_UNIFORM && type_ != fit::EXPONENTIAL
            && type_ != fit::TRIANGULAR && type_ != fit::
                ↪ LOGNORMAL)
        scale_y = y_mean_hist/y_mean_distrib;
    else
        scale_y = n_samples*width_bin;

    // if discrete distribution, add one point for each
        ↪ integer
    double step = (max_ - min_)/(double)n_points;
    if(distribution_->get_samples_type() == fit::sample_type
        ↪ ::DISCRETE)
    {
        step = 1;
        n_points++;
    }

    // define all scaled and offsetted points
    double x = min_;
    QList<QPointF> points;
    for(int i = 0; i < n_points; i++)
    {
```

```
            // scale and offset point
            double y_scaled = scale_y * distribution_->
                ↪ probability_function(x);
            double x_scaled = offset_ + scale_factor_*x;

            points.push_back(QPointF(x_scaled, y_scaled));

            x += step;
        }
        return points;
}

// #### GETTERS ####

QSharedPointer<fit::base_distrib> DistributionFit::
    ↪ getDistribution() const
{
    return distribution_;
}

double DistributionFit::getMin() const
{
    return min_;
}

double DistributionFit::getMax() const
{
    return max_;
}

QPair<fit::chisquare_test_result, fit::freq_table>
    ↪ DistributionFit::getChiSquare() const
{
    return chi_square_;
}

QList<double> DistributionFit::getSamples() const
{
    return samples_;
}

fit::sample_type DistributionFit::getSamplesType()
{
    return distribution_->get_samples_type();
}

QString DistributionFit::getName() const
{
    return QString::fromStdString(distribution_->name());
}

double DistributionFit::getOffset() const
```

```
{
    return offset_;
}

QVector<QPair<QString, double> > DistributionFit::getParams
    ↪ () const
{
    return params_;
}

double DistributionFit::getScaleFactor() const
{
    return scale_factor_;
}

// #### SETTERS ####

void DistributionFit::setMin(double min)
{
    min_ = min;
}

void DistributionFit::setMax(double max)
{
    max_ = max;
}

void DistributionFit::setChiSquare(const QPair<fit::
    ↪ chisquare_test_result, fit::freq_table> &chi_square)
{
    chi_square_ = chi_square;
}

void DistributionFit::setSamples(const QList<double> &
    ↪ samples)
{
    samples_ = samples;
}

void DistributionFit::setOffset(double offset)
{
    offset_ = offset;
}

void DistributionFit::setScaleFactor(double scale_factor)
{
    scale_factor_ = scale_factor;
}
```

Listagem C.11: distribution_fit.h

```
#ifndef DISTRIBUTIONFIT_H
```

```
#define DISTRIBUTIONFIT_H

#include <QString>
#include <QVector>
#include <QPointF>
#include <QPair>
#include <QSharedPointer>
#include "fit.h"

#include "defs.h"

class DistributionFit
{
public:
    DistributionFit(fit::distrib_ptr distribution, fit::
        ↪ _distribution_type type,
                    double offset, double scale_factor,
                        ↪ QList<double> samples);

    QList<QPointF> getProbabilityFunctionPoints(int n_points
        ↪ , double y_mean_hist, double width_bin, double
        ↪ n_samples);
    QSharedPointer<fit::base_distrib> getDistribution()
        ↪ const;

    // GETTERS
    QString getName() const;
    double getScaleFactor() const;
    double getOffset() const;
    QVector<QPair<QString, double> > getParams() const;
    QPair<fit::chisquare_test_result, fit::freq_table>
        ↪ getChiSquare() const;
    QList<double> getSamples() const;
    fit::sample_type getSamplesType();
    double getMin() const;
    double getMax() const;

    // SETTERS
    void setName(const QString &name);
    void setScaleFactor(double scale_factor);
    void setOffset(double offset);
    void setChiSquare(const QPair<fit::chisquare_test_result
        ↪ , fit::freq_table> &chi_square);
    void setSamples(const QList<double> &samples);
    void setMin(double min);
    void setMax(double max);

private:
    QSharedPointer<fit::base_distrib> distribution_;
    fit::_distribution_type type_;
    QVector<QPair<QString, double> > params_;
    double min_, max_;
```

```
    // scaled data
    QList<double> samples_;
    double offset_;
    double scale_factor_;

    // fit results
    QPair<fit::chisquare_test_result, fit::freq_table>
        ↪ chi_square_;
};

#endif // DISTRIBUTIONFIT_H
```

Listagem C.12: expo_distrib.cpp

```
//
    ↪ ------------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪

#include <distribution/expo_distrib.h>
#include <cmath>
```

```
#include <common/fit_utils.hpp>

using namespace fit;
using std::exp;
using std::log;

expo_distrib::expo_distrib(const data_set& values)
{
    _samples_type = sample_type::CONTINUOUS;

    _parameters.resize(PARAMS_NUMBER);
    set_params_mle(values);
}

expo_distrib::expo_distrib(const params_list& params)
{
    _samples_type = sample_type::CONTINUOUS;

    if (!check_parameters(params))
        throw distribution_exception("Invalid parameters on
            ↪ exponential distribution.");

    _parameters = params;
}

base_distrib *expo_distrib::clone()
{
    return new expo_distrib(*this);
}

std::string expo_distrib::name() const
{
    return "Exponential";
}

std::vector<std::string> expo_distrib::paramNames() const
{
    std::vector<std::string> paramNames = std::vector<std::
        ↪ string>();
    paramNames.push_back("Beta");

    return paramNames;
}

data_type expo_distrib::get_mean() const
{
    return _parameters[BETA];
}

data_type expo_distrib::get_variance() const
{
    data_type beta = _parameters[BETA];
```

```
    return beta * beta;
}

data_type expo_distrib::cumulative_function(data_type number
    ↪ ) const
{
    if (number < 0)
        return 0;

    return 1 - (exp( (number * -1) / _parameters[BETA]) );
}

data_type expo_distrib::probability_function(data_type
    ↪ number) const
{
    if (number < 0)
        return 0;

    data_type beta = _parameters[BETA];

    return (1 / beta) * exp((-number) / beta);
}

void expo_distrib::set_params_mle(const data_set& values)
{
    if (!check_values(values))
        throw distribution_exception("Invalid value on
            ↪ exponencial distribution.");

    _parameters[BETA] = utils::_mean(values);
}

data_type expo_distrib::get_random() const
{
    return (-_parameters[BETA]) * log(1 - random_number());
}

bool expo_distrib::check_parameters(const params_list&
    ↪ params) const
{
    return ((params.size() == PARAMS_NUMBER) && (params.at(
        ↪ BETA) > 0));
}
```

Listagem C.13: expo_distrib.h

```
//
    ↪ ------------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
```

```
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪

#ifndef EXPO_DISTRIB_H_
#define EXPO_DISTRIB_H_

/*
 * Exponential Distribution
 */

#include "base_distrib.h"

namespace fit
{
class expo_distrib : public base_distrib
{
protected:
    inline bool in_range(data_type value) const { return
        ↪ value >= 0; }

public:
    enum _expo_params
    {
        BETA,
        PARAMS_NUMBER
    };
```

```
    expo_distrib(const data_set& values); //gets
        ↪ distribution parameters from MLE
    expo_distrib(const params_list& params); //gets
        ↪ distribution parameters from params.

    //Return a clone of de distribuition allocated on heap
    base_distrib *clone();

    std::string name() const;
    std::vector<std::string> paramNames() const;

    ~expo_distrib() {};

    data_type get_mean() const;
    data_type get_variance() const;
    data_type get_mode() const { return 0; }

    data_type cumulative_function(data_type number) const;
        ↪ //cumulative distribution function
    data_type probability_function(data_type number) const;
        ↪ //probability density function

    /* Sets distribution parameters through maximum
        ↪ likelihood procedures */
    void set_params_mle(const data_set& values);

    /* Gets a random value according to distribution */
    data_type get_random() const;

    bool check_parameters(const params_list& params) const;
};
}

#endif /*EXPO_DISTRIB_H_*/
```

Listagem C.14: file_handler.cpp

```
#include "utils/file_handler.h"

#include <QFile>
#include "xlsxdocument.h"
#include <qwt_plot_renderer.h>

#include "defs.h"

FileHandler::FileHandler()
{

}

QList<double> FileHandler::readSamplesFromXlsx(const QString
    ↪ &file)
```

```
{
    QList<double> samples;

    // reading data
    QXlsx::Document xlsx(file);
    QXlsx::CellRange range =  xlsx.dimension();
    for(int i = range.firstRow(); i < range.lastRow()+1; i
        ↪ ++)
    {
        if(QXlsx::Cell *cell=xlsx.cellAt(i, 1))
        {
            if(cell->value().type() == QVariant::Double){
                samples << cell->value().toDouble();
            }
        }
    }
    if(samples.isEmpty())
        throw file_exception("First column of xlsx file is
            ↪ empty or file is invalid.");

    return samples;
}

QList<double> FileHandler::readSamplesFromTxt(const QString
    ↪ &file_name)
{
    QList<double> samples;

    QFile file(file_name);
    if (!file.open(QIODevice::ReadOnly | QIODevice::Text))
        return samples;

    QTextStream in(&file);
    while (!in.atEnd()) {
        QString line = in.readLine();
        bool ok;
        double value = line.toDouble(&ok);
        if(ok)
            samples.push_back(value);
    }
    if(samples.isEmpty())
        throw file_exception("No data could be read from
            ↪ file.");
    return samples;
}

void FileHandler::writeSamplesToTxt(const QList<double>
    ↪ samples, const QString &file)
{
    QFile g1f;
    g1f.setFileName(file);
    if(!g1f.open(QFile::WriteOnly | QFile::Text))
```

```
        return;

    QTextStream df(&g1f);
    foreach(double sample, samples)
        df << QString::number(sample) << "\n";

    g1f.flush();
    g1f.close();
}

void FileHandler::writeSamplesToXlsx(const QList<double>
    ↪ samples, const QString &file)
{
    QXlsx::Document xlsx;
    int i = 1;
    foreach(double sample, samples)
    {
        QString cell_index = "A" + QString::number(i++);
        xlsx.write(cell_index, sample);
    }
    xlsx.saveAs(file);
}

void FileHandler::exportGraph(QwtPlot *plot, const QString
    ↪ file)
{
    QwtPlotRenderer renderer;
    renderer.renderDocument(plot, file, plot->size());
}
```

Listagem C.15: file_handler.h

```
#ifndef FILEHANDLER_H
#define FILEHANDLER_H

#include <qwt_plot.h>
#include <QString>

class FileHandler
{
public:
    FileHandler();
    static QList<double> readSamplesFromXlsx(const QString &
        ↪ file);
    static QList<double> readSamplesFromTxt(const QString &
        ↪ file);
    static void writeSamplesToTxt(const QList<double>
        ↪ samples, const QString &file);
    static void writeSamplesToXlsx(const QList<double>
        ↪ samples, const QString &file);
    static void exportGraph(QwtPlot *plot, const QString
        ↪ file);
```

```
};

#endif // FILEHANDLER_H
```

Listagem C.16: fit.cpp

```
//
    ↪ ------------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪

#include <fit.h>
#include <common/fit_utils.hpp>

fit::distrib_ptr fit::get_distribution(const data_set&
    ↪ values, int type)
{
    base_distrib* distrib_pointer = 0;

    switch (type)
    {
        case DISCRETE_UNIFORM:
```

```
            distrib_pointer = new discrete_uniform_distrib(
                ↪ values);
            break;
        case POISSON:
            distrib_pointer = new poisson_distrib(values);
            break;
        case BETA:
            distrib_pointer = new beta_distrib(values);
            break;
        case EXPONENTIAL:
            distrib_pointer = new expo_distrib(values);
            break;
        case GAMMA:
            distrib_pointer = new gamm_distrib(values);
            break;
        case LOGNORMAL:
            distrib_pointer = new lognorm_distrib(values);
            break;
        case NORMAL:
            distrib_pointer = new norm_distrib(values);
            break;
        case TRIANGULAR:
            distrib_pointer = new tria_distrib(values);
            break;
        case UNIFORM:
            distrib_pointer = new unif_distrib(values);
            break;
        case WEIBULL:
            distrib_pointer = new weib_distrib(values);
            break;
        default:
            throw distribution_exception("Distribution not
                ↪ found");
            break;
    }

    return distrib_ptr(distrib_pointer);
}

fit::distrib_ptr fit::get_distribution(const params_list&
    ↪ params, int type)
{
    base_distrib* distrib_pointer = 0;

    switch (type)
    {
        case DISCRETE_UNIFORM:
            distrib_pointer = new discrete_uniform_distrib(
                ↪ params);
            break;
        case POISSON:
            distrib_pointer = new poisson_distrib(params);
```

```
            break;
        case BETA:
            distrib_pointer = new beta_distrib(params);
            break;
        case EXPONENTIAL:
            distrib_pointer = new expo_distrib(params);
            break;
        case GAMMA:
            distrib_pointer = new gamm_distrib(params);
            break;
        case LOGNORMAL:
            distrib_pointer = new lognorm_distrib(params);
            break;
        case NORMAL:
            distrib_pointer = new norm_distrib(params);
            break;
        case TRIANGULAR:
            distrib_pointer = new tria_distrib(params);
            break;
        case UNIFORM:
            distrib_pointer = new unif_distrib(params);
            break;
        case WEIBULL:
            distrib_pointer = new weib_distrib(params);
            break;
        default:
            throw distribution_exception("Distribution not
                ↪ found");
            break;
    }

    return distrib_ptr(distrib_pointer);
}

fit::distrib_ptr fit::get_default_distribution(int type)
{
    base_distrib* distrib_pointer = 0;

    params_list params = params_list();
    params.push_back(1);

    switch (type)
    {
        case DISCRETE_UNIFORM:
            params.push_back(2);
            distrib_pointer = new discrete_uniform_distrib(
                ↪ params);
            break;
        case POISSON:
            distrib_pointer = new poisson_distrib(params);
            break;
        case BETA:
```

```
            params.push_back(2);
            distrib_pointer = new beta_distrib(params);
            break;
        case EXPONENTIAL:
            distrib_pointer = new expo_distrib(params);
            break;
        case GAMMA:
            params.push_back(2);
            distrib_pointer = new gamm_distrib(params);
            break;
        case LOGNORMAL:
            params.push_back(2);
            distrib_pointer = new lognorm_distrib(params);
            break;
        case NORMAL:
            params.push_back(2);
            distrib_pointer = new norm_distrib(params);
            break;
        case TRIANGULAR:
            params.push_back(2);
            params.push_back(3);
            distrib_pointer = new tria_distrib(params);
            break;
        case UNIFORM:
            params.push_back(2);
            distrib_pointer = new unif_distrib(params);
            break;
        case WEIBULL:
            params.push_back(2);
            distrib_pointer = new weib_distrib(params);
            break;
        default:
            throw distribution_exception("Distribution not
                ↪ found");
            break;
    }

    return distrib_ptr(distrib_pointer);
}

fit::freq_table fit::get_freq_table(const data_set& values,
    ↪ int dist_type, int criteria, unsigned int max_bins)
{
    distrib_ptr dist = get_distribution(values, dist_type);
    return freq_table(values, *dist, (_bin_criteria)
        ↪ criteria, max_bins);
}

fit::freq_table fit::get_freq_table(const data_set& values,
    ↪ const base_distrib& distrib, int criteria, unsigned
    ↪ int max_bins)
{
```

```
    return freq_table(values, distrib, (_bin_criteria)
        ↪ criteria, max_bins);
}

fit::chisquare_test_result fit::get_chisquare_test(const
    ↪ freq_table& table)
{
    chisquare_test_result result = {0, 0, 0};

    result.chi_statistic = utils::_chisquare_statistic(table
        ↪ );
    result.degrees_freedom = table.get_df();
    result.chi_p_value = utils::_chisquare_pf(result.
        ↪ chi_statistic, result.degrees_freedom);

    return result;
}
```

Listagem C.17: fit.h

```
//
    ↪ ------------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
```

```
    ↪

#ifndef FIT_H_
#define FIT_H_

#include <memory>

#include "distribution/discrete_uniform_distrib.h"
#include "distribution/poisson_distrib.h"
#include "distribution/beta_distrib.h"
#include "distribution/expo_distrib.h"
#include "distribution/lognorm_distrib.h"
#include "distribution/tria_distrib.h"
#include "distribution/unif_distrib.h"
#include "distribution/weib_distrib.h"

#include "freq_table.h"

namespace fit
{
    enum _distribution_type
    {
        DISCRETE_UNIFORM,
        POISSON,
        BETA,
        EXPONENTIAL,
        GAMMA,
        LOGNORMAL,
        NORMAL,
        TRIANGULAR,
        UNIFORM,
        WEIBULL
    };

    struct chisquare_test_result
    {
        data_type chi_statistic; //the test statistic
        unsigned int degrees_freedom; //the number of
            ↪ degrees of freedom

        /* The significance probability of the test.
         * A small value of chi_p_value indicates a
         * significant difference between the data set
         * and the choosen distribution.
         */
        data_type chi_p_value;
    };

    typedef std::auto_ptr<base_distrib> distrib_ptr;

    distrib_ptr get_distribution(const data_set& values, int
        ↪  dist_type);
```

```
    distrib_ptr get_distribution(const params_list& params,
        ↪ int dist_type);
    distrib_ptr get_default_distribution(int dist_type);

    freq_table get_freq_table(const data_set& values, int
        ↪ dist_type, int criteria, unsigned int max_bins =
        ↪ _MAX_BINS);
    freq_table get_freq_table(const data_set& values, const
        ↪ base_distrib& distrib, int criteria, unsigned int
        ↪  max_bins = _MAX_BINS);

    chisquare_test_result get_chisquare_test(const
        ↪ freq_table& table);
}

#endif /*FIT_H_*/
```

Listagem C.18: fit_defs.h

```
//
    ↪ ------------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪
```

```
#ifndef FIT_DEFS_H_
#define FIT_DEFS_H_

#include <limits>
#include <list>
#include <string>

namespace fit
{
    /* Defines the precision */
    typedef double                      data_type;
    typedef std::list<data_type>        data_set;

    /* Frequency table constraint */
    enum _bin_criteria
    {
        STURGES,
        SQUAREROOT,
        OTHER
    };
    /* samples type */
    enum sample_type
    {
        DISCRETE,
        CONTINUOUS
    };
    const unsigned int _MAX_BINS = 40;

    /* Exceptions */
    struct value_exception
    {
        std::string msg;
        value_exception(std::string _msg) : msg(_msg) {}
    };

    struct freq_table_exception
    {
        std::string msg;
        freq_table_exception(std::string _msg) : msg(_msg)
            ↪ {}
    };

    struct distribution_exception
    {
        std::string msg;
        distribution_exception(std::string _msg) : msg(_msg)
            ↪ {}
    };
};

#endif /*FIT_DEFS_H_*/
```

Listagem C.19: fit_utils.cpp

```
//
    ↪ ------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------
    ↪

#include <common/fit_utils.hpp>
#include <cmath>

using namespace fit;

unsigned int fit::utils::_bin_width(_bin_criteria _criteria,
    ↪  int _samples_number, unsigned int _max_bins) {
    if (_samples_number <= 0)
        return 0;

    int ret = 0;
    switch (_criteria) {
    case SQUAREROOT:
        ret = std::floor (std::sqrt((double)_samples_number)
            ↪  + 0.5);
        break;
```

```
    case STURGES:
        ret = std::floor (1.5 + 3.222 * std::log10((double)
            ↪ _samples_number));
        break;
    case OTHER:
    default:
        return _max_bins;
    }

    return ret < _max_bins ? ret : _max_bins;
}

sample_type utils::_check_samples_type(data_set _values)
{
    double intpart;
    data_set::const_iterator it = _values.begin();
    while(it != _values.end() && std::modf((*it), &intpart)
        ↪ == 0.0)
        it++;

    if(it == _values.end())
        return sample_type::DISCRETE;
    return sample_type::CONTINUOUS;
}

data_type fit::utils::_chisquare_statistic(const freq_table&
    ↪  _table)
{
    if (_table.size() <= 5)
        throw freq_table_exception("Chi-Square test is not
            ↪ applicable on this frequency table. Number of
            ↪  intervals <= 5");

    data_type obs_freq, exp_freq;
    data_type result = 0.0;
    for (freq_table_t::const_iterator i = _table.begin(); i
        ↪ != _table.end(); i++)
    {
        obs_freq = i->get_observed_freq();
        exp_freq = i->get_expected_freq();
        result += std::pow(obs_freq - exp_freq, 2) /
            ↪ exp_freq;
    }
    return result;
}

data_type fit::utils::_chisquare_pf(data_type chisquare_stat
    ↪ , unsigned int df)
{
    return gammq(df / 2.0, chisquare_stat / 2.0);
}
```

```
//Returns de value of ln[Î“(xx)] for xx > 0.
data_type fit::utils::gammln(data_type xx)
{
    data_type x, y, tmp, ser;
    data_type cof[6] = { 76.18009172947146,
        ↪ -86.50532032941677,
            24.01409824083091, -1.231739572450155,
                ↪ 0.1208650973866179e-2,
            -0.5395239384953e-5 };
    int j;
    y = x = xx;
    tmp = x + 5.5;
    tmp -= (x + 0.5) * log (tmp);
    ser = 1.000000000190015;
    for (j = 0; j <= 5; j++)
        ser += cof[j] / ++y;
    return -tmp + log (2.5066282746310005 * ser / x);
}

void fit::utils::gcf(data_type *gammcf, data_type a,
    ↪ data_type x,
        data_type *gln)
/*Returns the incomplete gamma function Q(a, x) evaluated by
    ↪  its continued fraction represen-
 tation as gammcf. Also returns ln Î“(a) as gln.*/
{
    int i;
    data_type an, b, c, d, del, h;
    *gln = gammln (a);
    b = x + 1.0- a;
    c = 1.0 / DATAMIN;
    d = 1.0 / b;
    h = d;
    for (i = 1; i <= ITMAX; i++) {
        an = -i * (i - a);
        b += 2.0;
        d = an * d + b;
        if (fabs (d) < DATAMIN)
            d = DATAMIN;
        c = b + an / c;
        if (fabs (c) < DATAMIN)
            c = DATAMIN;
        d = 1.0 / d;
        del = d * c;
        h *= del;
        if (fabs (del - 1.0) < EPS)
            break;
    }
    if (i > ITMAX)
        throw value_exception("a too large, ITMAX too small
            ↪ in gcf");
    *gammcf = exp (-x + a * log (x) - (*gln)) * h;
```

```
}

void fit::utils::gser(data_type *gamser, data_type a,
    ↪ data_type x,
        data_type *gln) {
    /*Returns the incomplete gamma function P (a, x)
        ↪ evaluated by its series representation as gamser.
     Also returns ln Î“(a) as gln.*/

    int n;
    data_type sum, del, ap;
    *gln = gammln (a);
    if (x <= 0.0) {
        if (x < 0.0)
            throw value_exception("x less than 0 in routine
                ↪ gser");
        *gamser = 0.0;
        return;
    } else {
        ap = a;
        del = sum = 1.0 / a;
        for (n = 1; n <= ITMAX; n++) {
            ++ap;
            del *= x / ap;
            sum += del;
            if (fabs (del) < fabs (sum) * EPS) {
                *gamser =sum * exp (-x + a * log (x) - (*gln
                    ↪ ));
                return;
            }
        }
        throw value_exception("a too large, ITMAX too small
            ↪ in routine gser");
        return;
    }
}

data_type fit::utils::gammp(data_type a, data_type x)
//Returns the incomplete gamma function P (a, x).
{
    data_type gamser, gammcf, gln;
    if (x < 0.0|| a <= 0.0)
        throw value_exception("Invalid arguments in routine
            ↪ gammp");
    if (x < (a+1.0)) {
        gser(&gamser, a, x, &gln);
        return gamser;
    } else {
        gcf(&gammcf, a, x, &gln);
        return 1.0-gammcf;
    }
}
```

```
data_type fit::utils::gammq(data_type a, data_type x)
//Returns the incomplete gamma function Q(a, x) = 1 â^' P (a
    ↪ , x).
{
    data_type gamser, gammcf, gln;
    if (x < 0.0|| a <= 0.0)
        throw value_exception("Invalid arguments in routine
            ↪ gammq");
    if (x < (a + 1.0)) {
        gser (&gamser, a, x, &gln);
        return 1.0 - gamser;
    } else {
        gcf (&gammcf, a, x, &gln);
        return gammcf;
    }
}

//
    ↪ ==========================================================
    ↪

data_type fit::utils::betai(data_type a, data_type b,
    ↪ data_type x)
//Returns the incomplete beta function Ix (a, b).
{
    data_type bt;
    if (x < 0.0 || x > 1.0)
        throw value_exception("Bad x in routine betai: ");

    if (x == 0.0 || x == 1.0)
    bt=0.0;
    else
    bt=exp(gammln(a+b)-gammln(a)-gammln(b)+a*log(x)+b*log
        ↪ (1.0-x));
    if (x < (a+1.0)/(a+b+2.0))
    return bt*betacf(a,b,x)/a;
    else
    return 1.0-bt*betacf(b,a,1.0-x)/b;
}

data_type fit::utils::betacf(data_type a, data_type b,
    ↪ data_type x) {
    int m, m2;
    float aa, c, d, del, h, qab, qam, qap;
    qab=a+b;
    qap=a+1.0;
    qam=a-1.0;
    c=1.0;
    d=1.0-qab*x/qap;
    if (fabs(d) < DATAMIN)
        d=DATAMIN;
```

```
    d=1.0/d;
    h=d;
    for (m=1; m<=ITMAX; m++) {
        m2=2*m;
        aa=m*(b-m)*x/((qam+m2)*(a+m2));
        d=1.0+aa*d;
        if (fabs(d) < DATAMIN)
            d=DATAMIN;
        c=1.0+aa/c;
        if (fabs(c) < DATAMIN)
            c=DATAMIN;
        d=1.0/d;
        h *= d*c;
        aa = -(a+m)*(qab+m)*x/((a+m2)*(qap+m2));
        d=1.0+aa*d;
        if (fabs(d) < DATAMIN)
            d=DATAMIN;
        c=1.0+aa/c;
        if (fabs(c) < DATAMIN)
            c=DATAMIN;
        d=1.0/d;
        del=d*c;
        h *= del;

        if (fabs(del-1.0) < EPS)
            break;
    }
    if (m > ITMAX)
        throw value_exception("a or b too big, or MAXIT too
            ↪ small in betacf");

    return h;
}
```

Listagem C.20: fit_utils.hpp

```
//
    ↪ ------------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
```

```
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪

#ifndef FIT_UTILS_HPP_
#define FIT_UTILS_HPP_

#include "fit_defs.h"
#include "freq_table.h"

#include <cmath>
#include <list>

namespace fit
{
    namespace utils
    {
        const int ITMAX = 100;
        const float EPS = 3.0e-7;

#ifndef BORLAND
        const data_type DATAMIN = std::numeric_limits<
            ↪ data_type>::denorm_min();
#else
        const data_type DATAMIN = 1.0e-30; // It's approx.
            ↪ float min. No problem.
#endif

        const long double PI =
            ↪ 3.14159265358979323846264338327950288419 68;

        // check if samples are discrete or continuous
        sample_type _check_samples_type(data_set _values);

        //extern "C"
        //{
        /* Gamma utils. From Numerical Recipes in C, 1992.
            ↪ */
```

```
data_type gammln(data_type xx);
void gcf(data_type *gammcf, data_type a, data_type x
    ↪ , data_type *gln);
void gser(data_type *gamser, data_type a, data_type
    ↪ x, data_type *gln);
data_type gammp(data_type a, data_type x);
data_type gammq(data_type a, data_type x);

/* Beta utils. From Numerical Recipes in C, 1992. */
data_type betacf(data_type a, data_type b, data_type
    ↪  x);
data_type betai(data_type a, data_type b, data_type
    ↪ x);
//}

/*
 * SQUAREROOT is the square root os _samples_number
 * STURGES is 1.5 + 3.222 * log10(_samples_number)
 * The maximum number os bins is _max_bin.
 */
unsigned int _bin_width (_bin_criteria _criteria,
    ↪ int _samples_number, unsigned int _max_bins =
    ↪  _MAX_BINS);

data_type _chisquare_statistic(const freq_table&
    ↪ _table);
data_type _chisquare_pf(data_type chisquare_stat,
    ↪ unsigned int df); //chi-square probability
    ↪ function. Is an incomplete gamma function.

/* Statistical utilities */
template < class T > T _mean (const typename std::
    ↪ list < T >& values)
{
    if (values.empty ())
        return 0;
    T sum = 0;
    for (typename std::list < T >::const_iterator i
        ↪ = values.begin ();
         i != values.end (); i++)
    {
        sum += *i;
    }
    return sum / values.size ();
}

template < class T > T _variance (const std::list <
    ↪ T >& values)
{
    if (values.empty ())
        return 0;
    T temp = 0;
```

```
            T mean_value = mean (values);
            for (typename std::list < T >::const_iterator i
                ↪ = values.begin ();
                 i != values.end (); i++)
            {
                temp += (*i - mean_value) * (*i - mean_value
                    ↪ );
            }
            return temp / (values.size () - 1);
        }

        template < class T > T _variance (T mean, const std
            ↪ ::list < T >& values)
        {
            T temp = 0;
            for (typename std::list < T >::const_iterator i
                ↪ = values.begin ();
                 i != values.end (); i++)
            {
                temp += (*i - mean) * (*i - mean);
            }
            return temp / (values.size () - 1);
        }

        template < class T > T _std_dev (const std::list < T
            ↪ >& values)
        {
            if (values.empty ())
                return 0;
            T var_value = _variance (values);
            return std::sqrt (var_value);
        }
    };
};

#endif /*FIT_UTILS_HPP_*/
```

Listagem C.21: freq_table.cpp

```
//
    ↪ ------------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
```

```
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪

#include <freq_table.h>
#include <common/fit_utils.hpp>
#include <cmath>
#include <algorithm>

using namespace fit;

freq_table::freq_table()
{

}

freq_table::freq_table(const data_set& values, const
    ↪ base_distrib& distrib, _bin_criteria criteria,
    ↪ unsigned int max_bins)
    : _values(values)
{
    init_members(distrib, utils::_bin_width(criteria,
        ↪ _values.size(), max_bins));
    optimize(distrib);
    df = size() - distrib.get_params_number() - 1;
}

freq_table::freq_table(const data_set& values, const
    ↪ base_distrib& distrib, unsigned int intervals)
    : _values(values)
{
    init_members(distrib, intervals);
    optimize(distrib);
    df = size() - distrib.get_params_number() - 1;
}
```

```
void freq_table::init_members(const base_distrib& distrib,
    ↪ unsigned int intervals)
{
    if (_values.empty())
        throw freq_table_exception("No values on frequency
            ↪ table construction");

    if (intervals <= 0)
        throw freq_table_exception("Negative number of
            ↪ intervals on frequency table construction :"
            ↪ + intervals);

    data_type minor = *min_element(_values.begin(), _values.
        ↪ end());
    data_type major = *max_element(_values.begin(), _values.
        ↪ end());
    //data_type minor = floor(*min_element(_values.begin(),
        ↪ _values.end()));
    //data_type major = ceil(*max_element(_values.begin(),
        ↪ _values.end()));

    data_type difference = major - minor;

    if (difference != difference) //std::isnan(difference)
        throw freq_table_exception("Constant values on
            ↪ frequency table construction");

    //data_type step = (difference * 1.01) / intervals; //
        ↪ Little work-around to work ok with float-point
        ↪ numbers.
    data_type step = difference / intervals; //Little work-
        ↪ around to work ok with float-point numbers.


    if(distrib.get_samples_type() == sample_type::DISCRETE)
        step = floor(step);
    if(step <= 0)
        step = 1;

    /*
     * intervals_values is a list where intervals
         ↪ correspondent
     * values are classified.
     */
    std::vector<data_set> intervals_values(intervals);

    /* classifies the values */
    for (data_set::const_iterator i = _values.begin (); i !=
        ↪  _values.end (); i++)
    {
        int auxtmp = std::floor( (*i - minor) / step );
```

```
            if (auxtmp >= intervals_values.size())
            {
                auxtmp = intervals_values.size() - 1;
            }
            intervals_values.at(auxtmp).push_back(*i);
        }

        bool done = false;
        for (unsigned int i = 0; i < intervals && done != true;
            ↪ i++)
        {
            data_type interv_min = minor + (i * step);
            data_type interv_max = minor + ((i * step) + step);

            if(interv_max > major)
            {
                interv_max = major;
                done = true;
            }
            data_type interv_exp_freq = distrib.expected_freq(
                ↪ interv_min, interv_max, _values.size());
            if(interv_exp_freq <= 0)
                continue;

            freq_table_entry new_entry(intervals_values.at(i),
                ↪ interv_min, interv_max, interv_exp_freq);

            push_back(new_entry);
        }
}

void freq_table::optimize(const base_distrib& distrib)
{
    //divide_intervals(distrib);
    merge_intervals();
}

void freq_table::divide_intervals(const base_distrib&
    ↪ distrib)
{
    for (freq_table_t::iterator it = begin(); it != end() ;
        ↪ )
    {
        if ( it->get_observed_freq() > (10.0 * (_values.size
            ↪ ()/size())) ) //Barbetta criteria.
        {
            data_type minor = it->get_min_value();
            data_type major = it->get_max_value();
            data_type medium = (minor + major) / 2.0;
            data_set actual_interval_v = it->
                ↪ get_entry_values();
            data_set previous_interval_v;
```

```
                data_set subsequent_interval_v;
                for (data_set::const_iterator itV =
                    ↪ actual_interval_v.begin(); itV !=
                    ↪ actual_interval_v.end(); itV++)
                {
                    if (*itV < medium)
                    {
                        previous_interval_v.push_back(*itV);
                    }
                    else
                    {
                        subsequent_interval_v.push_back(*itV);
                    }
                }
                if (previous_interval_v.size() == 0)
                //this happens when there are too many values
                    ↪ repeated.
                {
                    it++;
                }
                else
                {
                    freq_table_entry previous_interval(
                        ↪ previous_interval_v, minor, medium,
                        ↪ distrib.expected_freq(minor, medium,
                        ↪ _values.size()));
                    freq_table_entry subsequent_interval(
                        ↪ subsequent_interval_v, medium, major,
                        ↪  distrib.expected_freq(medium, major,
                        ↪  _values.size()));
                    it = erase(it);
                    it = insert(it,subsequent_interval);
                    it = insert(it,previous_interval);
                }
            }
            else { it++; }
        }
    }

    void freq_table::merge_intervals()
    {
        freq_table_t::iterator it_temp;
        freq_table_t::iterator end_it = end();
        end_it--;

        data_type expected_freq_sum = 0;
        for (freq_table_t::iterator it = begin(); it != end();
            ↪ it++)
        {
            expected_freq_sum += it->get_expected_freq();
        }
```

```
    if (expected_freq_sum < 5) return; //return if it is
        ↪ impossible to merge.

    //Merge entry whose expected frequency is less than 5
    for (freq_table_t::iterator it = begin() ; it != end() ;
        ↪ )
    {
        if (it->get_observed_freq() < 6)
        {
            if (it == end_it)
            {
                it_temp = it;
                it_temp--;
                it->merge(*it_temp);
                erase(it_temp);
            }
            else
            {
                it_temp = it;
                it_temp++;
                it_temp->merge(*it);
                it = erase(it);
            }
        }
        else
        {
            it++;
        }
    }
}

data_type freq_table::get_square_error() const
{
    data_type sum = 0;
    for (freq_table_t::const_iterator it = begin(); it !=
        ↪ end(); it++)
    {
        data_type expfreq = it->get_expected_freq();
        unsigned int obsfreq = it->get_observed_freq();
        sum += (expfreq - obsfreq) * (expfreq - obsfreq);
    }

    return std::sqrt(sum) / size();
}
```

Listagem C.22: freq_table.h

```
//
    ↪ ------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
```

```
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------
    ↪

#ifndef FREQ_TABLE_H_
#define FREQ_TABLE_H_

#include "common/freq_table_entry.h"
#include "distribution/base_distrib.h"

namespace fit
{
typedef std::list<freq_table_entry> freq_table_t; //
    ↪ frequency table type
class freq_table : public freq_table_t
{
private:
    data_set _values;
    unsigned int df; //degrees of freedom

    void init_members(const base_distrib& distrib, unsigned
        ↪ int intervals);
    void divide_intervals(const base_distrib& distrib);
    void merge_intervals();

protected:
```

```
    /*
     * This procedure divide information excess intervals
     * and merge whoses have expected frequency less than 5.
     * (See Chi-Square test requisits.)
     */
    void optimize(const base_distrib& distrib);

public:
    freq_table();
    freq_table(const data_set& values, const base_distrib&
        ↪ distrib, _bin_criteria criteria, unsigned int
        ↪ max_bins = _MAX_BINS);
    freq_table(const data_set& values, const base_distrib&
        ↪ distrib, unsigned int intervals);
    ~freq_table() {};

    data_type get_square_error() const;
    unsigned int get_df() const { return df; }
};
}

#endif /*FREQ_TABLE_H_*/
```

Listagem C.23: freq_table_entry.cpp

```
//
    ↪ ------------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
```

```
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------
    ↪

#include <common/freq_table_entry.h>

using namespace fit;

freq_table_entry::freq_table_entry(const data_set&
    ↪ entry_values, data_type min_value,
        data_type max_value, data_type expected_freq)
: _entry_values(entry_values), _expected_freq(expected_freq)
    ↪ ,
  _min_value(min_value), _max_value(max_value)
{
}

void freq_table_entry::set_expected_freq(data_type
    ↪ expected_freq)
{
    _expected_freq = expected_freq < 0 ? 0 : expected_freq;
}

data_type freq_table_entry::get_min_value() const
{
    return _min_value;
}

data_type freq_table_entry::get_max_value() const
{
    return _max_value;
}

unsigned int freq_table_entry::get_observed_freq() const
{
    return _entry_values.size();
}

data_type freq_table_entry::get_expected_freq() const
{
    return _expected_freq;
}

data_set freq_table_entry::get_entry_values() const
{
    return _entry_values;
}
```

```
void freq_table_entry::merge(const freq_table_entry&
    ↪ other_entry)
{
    data_set tmpset = other_entry.get_entry_values();
    _entry_values.merge(tmpset);

    _expected_freq += other_entry.get_expected_freq();
}
```

Listagem C.24: freq_table_entry.h

```
//
    ↪ ------------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪

#ifndef FREQ_TABLE_ENTRY_H_
#define FREQ_TABLE_ENTRY_H_

#include "fit_defs.h"

namespace fit
{
```

```
class freq_table_entry
{
private:
    data_set _entry_values;
    data_type _expected_freq; //distribution-function
        ↪ dependent

    /* This is not the min/max values of _entry_values
     * but the interval's limits
     */
    data_type _min_value;
    data_type _max_value;

public:
    freq_table_entry(const data_set& entry_values, data_type
        ↪  min_value,
            data_type max_value, data_type expected_freq =
                ↪ 0);

    ~freq_table_entry() {};

    data_type get_min_value() const;
    data_type get_max_value() const;
    unsigned int get_observed_freq() const;
    data_type get_expected_freq() const;
    data_set get_entry_values() const;

    void set_expected_freq(data_type expected_freq);

    void merge(const freq_table_entry& other_entry);
};
}

#endif /*FREQ_TABLE_ENTRY_H_*/
```

Listagem C.25: gamm_distrib.cpp

```
//
    ↪ ------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
```

```
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪

#include <distribution/gamm_distrib.h>
#include <common/fit_utils.hpp>
#include <cmath>

using namespace fit;
using std::exp;
using std::pow;
using std::log;

gamm_distrib::gamm_distrib(const data_set& values)
{
    _samples_type = sample_type::CONTINUOUS;

    _parameters.resize(PARAMS_NUMBER);
    set_params_mle(values);
}

gamm_distrib::gamm_distrib(const params_list& params)
{
    _samples_type = sample_type::CONTINUOUS;

    if (!check_parameters(params))
        throw distribution_exception("Invalid parameters on
            ↪ Gamma distribution.");

    _parameters = params;
}

base_distrib *gamm_distrib::clone()
{
    return new gamm_distrib(*this);
}
```

```
std::string gamm_distrib::name() const
{
    return "Gamma";
}

std::vector<std::string> gamm_distrib::paramNames() const
{
    std::vector<std::string> paramNames = std::vector<std::
        ↪ string>();
    paramNames.push_back("Alpha");
    paramNames.push_back("Beta");

    return paramNames;
}

data_type gamm_distrib::get_mean() const
{
    return _parameters[ALPHA] * _parameters[BETA];
}

data_type gamm_distrib::get_variance() const
{
    data_type beta = _parameters[BETA];
    return _parameters[ALPHA] * (beta * beta);
}

data_type gamm_distrib::get_mode() const
{
    data_type alpha = _parameters[ALPHA];
    if (alpha < 1)
        return 0;

    return _parameters[BETA] * (alpha - 1);
}

data_type gamm_distrib::cumulative_function(data_type number
    ↪ ) const
{
    return utils::gammp(_parameters[ALPHA], (number /
        ↪ _parameters[BETA]));
}

data_type gamm_distrib::probability_function(data_type
    ↪ number) const
{
    if (number <= 0)
        return 0;

    data_type alpha = _parameters[ALPHA];
    data_type beta = _parameters[BETA];

    data_type firstterm = pow(beta, -alpha) * pow(number,
```

```
        ↪ alpha - 1) * exp(-number / beta);
    return firstterm / exp(utils::gammln(alpha));
}

void gamm_distrib::set_params_mle(const data_set& values)
{
    if (!check_values(values))
        throw distribution_exception("Invalid value on Gamma
            ↪ distribution.");

    data_type mean = utils::_mean(values);
    data_type variance = utils::_variance(mean, values);

    _parameters[ALPHA] = (mean * mean) / variance;
    _parameters[BETA] = variance / mean;

    if (!check_parameters(_parameters))
        throw distribution_exception("Invalid parameters on
            ↪ Gamma distribution.");
}

data_type gamm_distrib::get_random() const
{
    data_type alpha = _parameters[ALPHA];
    data_type beta = _parameters[BETA];

    // according to Law and Kelton, Simulation Modeling and
        ↪ Analysis, 1991
    // pages 488-489
    if (alpha < 1.0)
    {
        data_type b = (exp(1.0) + alpha) / exp(1.0);
        int counter = 0;
        while (counter < 1000)
        {
            // step 1.
            data_type P = b * random_number();
            if (P <= 1.0){
                // step 2.
                data_type Y = pow(P, 1.0 / alpha);
                data_type U2 = random_number();
                if (U2 <= exp(-Y))
                    return beta * Y;
            }
            else
            {
                // step 3.
                data_type Y = -log((b - P) / alpha);
                data_type U2 = random_number();
                if (U2 <= pow(Y, alpha - 1.0))
                    return beta * Y;
            }
```

```
            counter++;
        }
        return 1.0;
    }
    else if (alpha > 1.0)
    {
        // according to Law and Kelton, Simulation Modeling
            ↪ and Analysis, 1991
        // pages 488-489
        data_type a = 1.0 / sqrt(2.0 * alpha - 1.0);
        data_type b = alpha - log(4.0);
        data_type q = alpha + (1.0 / a);
        data_type theta = 4.5;
        data_type d = 1.0 + log(theta);
        int counter = 0;
        while (counter < 1000){
            // step 1.
            data_type U1 = random_number();
            data_type U2 = random_number();
            // step 2.
            data_type V = a * log(U1 / (1.0 - U1));
            data_type Y = alpha * exp(V);
            data_type Z = U1 * U1 * U2;
            data_type W = b + q * V - Y;
            // step 3.
            if ((W + d - theta * Z) >= 0.0)
                return beta * Y;
            else{
                // step 4.
                if (W > log(Z))
                    return beta * Y;
            }
            counter++;
        }
        return 1.0;
    }
    else // alpha == 1.0
    {
        // Gamma(1.0, beta) ~ exponential with mean = beta
        return -beta * log(random_number());
    }
}

bool gamm_distrib::check_parameters(const params_list&
    ↪ params) const
{
    return ((params.size() == PARAMS_NUMBER) &&
            (params.at(ALPHA) > 0) &&
            (params.at(BETA) > 0));
}
```

Listagem C.26: gamm_distrib.h

```
//
    ↪ ------------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪

#ifndef GAMM_DISTRIB_H_
#define GAMM_DISTRIB_H_

/*
 * Gamma Distribution
 */

#include "base_distrib.h"

namespace fit
{
class gamm_distrib : public base_distrib
{
protected:
    inline bool in_range(data_type value) const { return
        ↪ value >= 0; }
```

```
public:
    enum _gamm_params
    {
        ALPHA,
        BETA,
        PARAMS_NUMBER
    };

    gamm_distrib(const data_set& values); //gets
        ↪ distribution parameters from MLE
    gamm_distrib(const params_list& params); //gets
        ↪ distribution parameters from params.

    //Return a clone of de distribuition allocated on heap
    base_distrib *clone();

    std::string name() const;
    std::vector<std::string> paramNames() const;

    ~gamm_distrib() {};

    data_type get_mean() const;
    data_type get_variance() const;
    data_type get_mode() const;

    data_type cumulative_function(data_type number) const;
        ↪ //cumulative distribution function
    data_type probability_function(data_type number) const;
        ↪ //probability density function

    /* Sets distribution parameters through maximum
        ↪ likelihood procedures */
    void set_params_mle(const data_set& values);

    /* Gets a random value according to distribution */
    data_type get_random() const;

    bool check_parameters(const params_list& params) const;
};
}

#endif /*GAMM_DISTRIB_H_*/
```

Listagem C.27: gof_tester.cpp

```
#include "model/gof_tester.h"

#include <QMessageBox>
#include "fit_defs.h"

#include "utils/utils.h"
```

```
#include "utils/graph_plot_factory.h"

GOFTester::GOFTester()
{

}

QList<double> GOFTester::getRandomSamples(const int
    ↪ numb_samples, const QVector<double> params,
                                              const int
                                                  ↪ distrib_type
                                                  ↪ )
{
    samples_ = GoodnessOfFit::generateRandomNumbers(
        ↪ numb_samples, params, distrib_type);
    return samples_;
}

// SETTERS

void GOFTester::setSamples(const QList<double> samples)
{
    std::list<double> temp_samples = samples.toStdList();
    temp_samples.sort();
    samples_ = QList<double>::fromStdList(temp_samples);
}

HistogramPlot* GOFTester::createHistogram(const int
    ↪ bin_criteria, const int num_bins)
{
    switch(bin_criteria)
    {
    case fit::SQUAREROOT:
        histogram_ = Histogram(samples_, fit::SQUAREROOT,
            ↪ fit::_MAX_BINS);
        break;
    case fit::STURGES:
        histogram_ = Histogram(samples_, fit::STURGES, fit::
            ↪ _MAX_BINS);
        break;
    case fit::OTHER:
        histogram_ = Histogram(samples_, fit::OTHER, fit::
            ↪ _MAX_BINS, num_bins);
        break;
    default:
        histogram_ = Histogram(samples_, fit::OTHER, fit::
            ↪ _MAX_BINS, fit::_MAX_BINS);
        break;
    }

    return GraphPlotFactory::createHistogramPlot(histogram_.
        ↪ getBins(), histogram_.getHighestFrequency());
```

```
}

QList<QPair<QString, QPair<int, double> > > GOFTester::
    ↪ getHistogramInfo()
{
    return Utils::convertHistToList(histogram_);
}

void GOFTester::fitDistributions(QList<int> indexes_distrib,
    ↪  int bin_criteria)
{
    indexes_distributions_ = indexes_distrib;

    // create distributions
    QStringList distrib_error_fit =
        ↪ createSelectedDistributions();

    // fit distributions
    distrib_error_fit += GoodnessOfFit::fitDistributions(
        ↪ distributions_fit_, bin_criteria);

    Utils::sortDistributionsByChiStat(distributions_fit_);

    // plot distributions
    prepareAndPlotDistributions();
    emit sigShowDistributionsInfo(distributions_fit_,
        ↪ distrib_error_fit);
}

Histogram GOFTester::getHistogram() const
{
    return histogram_;
}

void GOFTester::setHistogram(const Histogram &histogram)
{
    histogram_ = histogram;
}

QList<int> GOFTester::getIndexes_distributions() const
{
    return indexes_distributions_;
}

void GOFTester::setIndexes_distributions(const QList<int> &
    ↪ indexes_distributions)
{
    indexes_distributions_ = indexes_distributions;
}

QList<DistributionFit *> GOFTester::getDistributions_fit()
    ↪ const
```

```
{
    return distributions_fit_;
}

void GOFTester::setDistributions_fit(const QList<
    ↪ DistributionFit *> &distributions_fit)
{
    distributions_fit_ = distributions_fit;
}

QList<double> GOFTester::getSamples() const
{
    return samples_;
}

QStringList GOFTester::createSelectedDistributions()
{
    QStringList distrib_error;
    distributions_fit_.clear();
    for(int i = 0; i < indexes_distributions_.size(); i++)
    {
        // get name
        fit::_distribution_type type = (fit::
            ↪ _distribution_type)indexes_distributions_.at(
            ↪ i);
        QString name = QString::fromStdString(fit::
            ↪ get_default_distribution(type)->name());

        try
        {
            DistributionFit *distrib =  GoodnessOfFit::
                ↪ createDistributionFit(samples_, type);

            // add distribution to its list
            distributions_fit_.push_back(distrib);
        }
        catch(...)
        {
            distrib_error.push_back(name);
            continue;
        }
    }
    return distrib_error;
}

void GOFTester::prepareAndPlotDistributions()
{
    QList<QColor> colors = Utils::getSortedColors(
        ↪ distributions_fit_.size());
    QList<QwtPlotCurve*> curves;

    double mean_y_hist = histogram_.getFrequencyAtMean();
```

```
int n_samples = samples_.size();
int i_color = 0;
foreach(DistributionFit* distrib, distributions_fit_)
{
    if(distrib->getSamplesType() == fit::sample_type::
        ↪ DISCRETE)
    {
        QList<QPointF> points = distrib->
            ↪ getProbabilityFunctionPoints((distrib->
            ↪ getMax() - distrib->getMin() + 1),

        curves.push_back(GraphPlotFactory::
            ↪ createMassFunctionPlot(distrib->getName()
            ↪ ,

    }else
    {
        QList<QPointF> points = distrib->
            ↪ getProbabilityFunctionPoints(2000,
            ↪ mean_y_hist,
```

```
                curves.push_back(GraphPlotFactory::
                    ↪ createDensityFunctionPlot(distrib->
                    ↪ getName(),
        }
    }
    emit sigPlotDistributions(curves);
}
```

Listagem C.28: gof_tester.h

```
#ifndef GOFTESTER_H
#define GOFTESTER_H

#include "goodness_of_fit.h"

#include <QObject>

#include "histogram.h"
#include "view/plot/histogram_plot.h"
#include "view/plot/density_function_plot.h"
#include "view/plot/mass_function_plot.h"
#include "model/distribution_fit.h"

class GOFTester: public QObject
{
    Q_OBJECT

public:
    GOFTester();

    HistogramPlot* createHistogram(const int bin_criteria,
        ↪ const int num_bins);
    QList<QPair<QString, QPair<int, double> > >
        ↪ getHistogramInfo();
    void fitDistributions(QList<int> indexes_distrib, int
        ↪ bin_criteria);
```

```
    // GETTERS
    QList<double> getSamples() const;
    Histogram getHistogram() const;
    QList<int> getIndexes_distributions() const;
    QList<DistributionFit *> getDistributions_fit() const;

    // SETTERS
    void setSamples(const QList<double> samples);
    void setHistogram(const Histogram &histogram);
    void setIndexes_distributions(const QList<int> &
        ↪ indexes_distributions);
    void setDistributions_fit(const QList<DistributionFit *>
        ↪  &distributions_fit);

private:
    QList<double> samples_;
    Histogram histogram_;
    QList<int> indexes_distributions_;
    QList<DistributionFit*> distributions_fit_;

    QStringList createSelectedDistributions();
    void prepareAndPlotDistributions();

public slots:
    QList<double> getRandomSamples(const int numb_samples,
        ↪ const QVector<double> params, const int
        ↪ distrib_type);

signals:
    void sigPlotDistributions(QList<QwtPlotCurve*> curves);
    void sigShowDistributionsInfo(const QList<
        ↪ DistributionFit*> distributions_fit, const
        ↪ QStringList distrib_error_fit);
};

#endif // GOFTESTER_H
```

Listagem C.29: GOFTester.pro

```
#-------------------------------------------------
#
# Project created by QtCreator 2016-10-05T23:43:22
#
#-------------------------------------------------

QT       += core gui

CONFIG += qwt

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = GOFTester
```

```
TEMPLATE = app

INCLUDEPATH += $$PWD/libs/lib_fit/includes \
               $$PWD/includes \
 #              /usr/local/qwt-6.1.4-svn/include
               #ubuntu notebook
               #/usr/local/qwt-6.1.3/include

#ubuntu trabalho
#LIBS += -L/usr/local/qwt-6.1.4-svn/lib -lqwt

#ubuntu notebook
#LIBS += -L/usr/local/qwt-6.1.3/lib -lqwt

include(libs/qtxlsx/src/xlsx/qtxlsx.pri)

FORMS += \
    forms/main_window.ui \
    forms/widget_chi_results.ui \
    forms/widget_distrib_to_fit.ui \
    forms/widget_distribution_parameters.ui \
    forms/widget_rand_number_generator.ui \
    forms/widget_samples_and_histogram.ui \
    forms/widget_selectable_distrib.ui

DISTFILES += \
    rsc/beta_graph.svg \
    rsc/disc_unif_graph.svg \
    rsc/expo_graph.svg \
    rsc/gamma_graph.svg \
    rsc/lognormal.svg \
    rsc/normal_graph.svg \
    rsc/poisson_graph.svg \
    rsc/triang_graph.svg \
    rsc/uniform.svg \
    rsc/weibull.svg

HEADERS += \
    includes/model/distribution_fit.h \
    includes/model/gof_tester.h \
    includes/model/goodness_of_fit.h \
    includes/model/histogram.h \
    includes/model/histogram_bin.h \
    includes/utils/file_handler.h \
    includes/utils/utils.h \
    includes/view/plot/density_function_plot.h \
    includes/view/plot/histogram_plot.h \
    includes/view/plot/mass_function_plot.h \
    includes/view/main_window.h \
    includes/view/widget_chi_results.h \
    includes/view/widget_distrib_to_fit.h \
    includes/view/widget_distribution_parameters.h \
```

```
    includes/view/widget_rand_number_generator.h \
    includes/view/widget_samples_and_histogram.h \
    includes/view/widget_selectable_distrib.h \
    includes/defs.h \
    includes/view/plot/graph_plot.h \
    libs/lib_fit/includes/common/fit_utils.hpp \
    libs/lib_fit/includes/common/freq_table_entry.h \
    libs/lib_fit/includes/distribution/base_distrib.h \
    libs/lib_fit/includes/distribution/beta_distrib.h \
    libs/lib_fit/includes/distribution/
        ↪ discrete_uniform_distrib.h \
    libs/lib_fit/includes/distribution/expo_distrib.h \
    libs/lib_fit/includes/distribution/gamm_distrib.h \
    libs/lib_fit/includes/distribution/lognorm_distrib.h \
    libs/lib_fit/includes/distribution/norm_distrib.h \
    libs/lib_fit/includes/distribution/poisson_distrib.h \
    libs/lib_fit/includes/distribution/tria_distrib.h \
    libs/lib_fit/includes/distribution/unif_distrib.h \
    libs/lib_fit/includes/distribution/weib_distrib.h \
    libs/lib_fit/includes/fit.h \
    libs/lib_fit/includes/fit_defs.h \
    libs/lib_fit/includes/freq_table.h \
    includes/utils/graph_plot_factory.h

SOURCES += \
    src/model/distribution_fit.cpp \
    src/model/gof_tester.cpp \
    src/model/goodness_of_fit.cpp \
    src/model/histogram.cpp \
    src/model/histogram_bin.cpp \
    src/utils/file_handler.cpp \
    src/utils/utils.cpp \
    src/view/main_window.cpp \
    src/view/widget_chi_results.cpp \
    src/view/widget_distrib_to_fit.cpp \
    src/view/widget_distribution_parameters.cpp \
    src/view/widget_rand_number_generator.cpp \
    src/view/widget_samples_and_histogram.cpp \
    src/view/widget_selectable_distrib.cpp \
    src/view/plot/density_function_plot.cpp \
    src/view/plot/graph_plot.cpp \
    src/view/plot/histogram_plot.cpp \
    src/view/plot/mass_function_plot.cpp \
    src/main.cpp \
    libs/lib_fit/src/common/fit_utils.cpp \
    libs/lib_fit/src/common/freq_table_entry.cpp \
    libs/lib_fit/src/distributions/base_distrib.cpp \
    libs/lib_fit/src/distributions/beta_distrib.cpp \
    libs/lib_fit/src/distributions/discrete_uniform_distrib.
        ↪ cpp \
    libs/lib_fit/src/distributions/expo_distrib.cpp \
    libs/lib_fit/src/distributions/gamm_distrib.cpp \
```

```
    libs/lib_fit/src/distributions/lognorm_distrib.cpp \
    libs/lib_fit/src/distributions/norm_distrib.cpp \
    libs/lib_fit/src/distributions/poisson_distrib.cpp \
    libs/lib_fit/src/distributions/tria_distrib.cpp \
    libs/lib_fit/src/distributions/unif_distrib.cpp \
    libs/lib_fit/src/distributions/weib_distrib.cpp \
    libs/lib_fit/src/fit.cpp \
    libs/lib_fit/src/freq_table.cpp \
    src/utils/graph_plot_factory.cpp

RESOURCES += \
    rsc/rsc.qrc
```

Listagem C.30: goodness_of_fit.cpp

```
#include "model/goodness_of_fit.h"

#include "fit_defs.h"
#include "common/fit_utils.hpp"

#include "utils/utils.h"

GoodnessOfFit::GoodnessOfFit()
{

}

QList<double> GoodnessOfFit::generateRandomNumbers(int
    ↪ num_samples, QVector<double> params, int distrib_type
    ↪ )
{
    QList<double> rand_numbers;

    fit::distrib_ptr distribution;
    try
    {
        distribution = fit::get_distribution(params.
            ↪ toStdVector(), distrib_type);
    }
    catch(...)
    {
        throw;
    }

    for(int i = 0; i < num_samples; i++)
        rand_numbers.push_back(distribution->get_random());

    return rand_numbers;
}

DistributionFit* GoodnessOfFit::createDistributionFit(QList<
    ↪ double> samples, fit::_distribution_type type)
```

```
{
    // get scaled and offsetted samples
    QPair<QPair<double, double>, QList<double> >
        ↪ scales_and_samples = Utils::scaleSamples(type,
        ↪ samples);

    // set scaled samples and scale and offset
    QPair<double, double> transformation =
        ↪ scales_and_samples.first;
    std::list<double> scaled_samples = scales_and_samples.
        ↪ second.toStdList();

    // create distribution fit
    fit::distrib_ptr distrib_pointer = fit::get_distribution
        ↪ (scaled_samples, type);
    return new DistributionFit(distrib_pointer, type,
        ↪ transformation.first,
                               transformation.second, QList<
                                   ↪ double>::fromStdList(
                                   ↪ scaled_samples));
}

QStringList GoodnessOfFit::fitDistributions(QList<
    ↪ DistributionFit *> &distributions, int bin_criteria)
{
    QStringList not_fitted_distributions; // stores
        ↪ disbritutions that failed to fit
    foreach(DistributionFit *distribution, distributions)
    {
        fit::distrib_ptr distrib_ptr = (fit::distrib_ptr)(
            ↪ distribution->getDistribution()->clone());
        fit::freq_table freq_table;
        fit::chisquare_test_result chi_result;

        try
        {
            // get frequency table
            freq_table = fit::get_freq_table(distribution->
                ↪ getSamples().toStdList(), *distrib_ptr,
                ↪ bin_criteria);

            // get chi results
            chi_result = fit::get_chisquare_test(freq_table)
                ↪ ;
        }
        catch(...)
        {
            not_fitted_distributions.push_back(distribution
                ↪ ->getName());
            distributions.removeOne(distribution);
            continue;
        }
```

```
            distribution->setChiSquare(QPair<fit::
                ↪ chisquare_test_result,fit::freq_table>(
                ↪ chi_result, freq_table));
        }
        return not_fitted_distributions;
}
```

Listagem C.31: goodness_of_fit.h

```
#ifndef GOODNESSOFFIT_H
#define GOODNESSOFFIT_H

#include <QList>
#include "fit.h"

#include "distribution_fit.h"

class GoodnessOfFit
{
public:
    GoodnessOfFit();
    static QList<double> generateRandomNumbers(int
        ↪ num_samples, QVector<double> params, int
        ↪ distrib_type);
    static DistributionFit* createDistributionFit(QList<
        ↪ double> samples, fit::_distribution_type type);
    static QStringList fitDistributions(QList<
        ↪ DistributionFit*> &distributions, int
        ↪ bin_criteria);
};

#endif // GOODNESSOFFIT_H
```

Listagem C.32: graph_plot.cpp

```
#include "view/plot/graph_plot.h"

#include <qwt_plot_canvas.h>
#include <qwt_legend.h>
#include <qwt_legend_label.h>
#include <qwt_plot_grid.h>

GraphPlot::GraphPlot( QWidget *parent ):
    QwtPlot( parent )
{
}

void GraphPlot::init()
{
    QwtPlotCanvas *canvas = new QwtPlotCanvas();
    canvas->setPalette( Qt::white );
```

```
    setCanvas( canvas );

    setAutoReplot( true );

    QwtPlotGrid *grid = new QwtPlotGrid;
    grid->enableX(false);
    grid->enableY(true);
    grid->enableXMin(false);
    grid->enableYMin(false);
    grid->setMajorPen( Qt::black, 0, Qt::DotLine );
    grid->attach(this);
}

void GraphPlot::plotHistogram(HistogramPlot *histogram)
{
    setAxisScale(QwtPlot::xBottom, histogram->getMin(),
        ↪ histogram->getMax());
    height_canvas_ = histogram->getHighestFrequency() * 1.2;

    histogram->attach(this);
}

void GraphPlot::plotProbalityFunctions(QList<QwtPlotCurve*>
    ↪ curves)
{
    foreach(QwtPlotCurve *curve, curves)
    {
        curve->attach(this);
        curve->hide();
    }
}

void GraphPlot::fitCanvasToHistogram(bool fit)
{
   if(fit)
       setAxisScale(QwtPlot::yLeft, 0, height_canvas_); //
           ↪ fixed y-axis size
   else
       setAxisAutoScale(QwtPlot::yLeft, true); // auto
           ↪ scaled y-axis
}

void GraphPlot::enableLegends(bool enable)
{
    if(enable)
    {
        QwtLegend *legend = new QwtLegend;
        legend->setDefaultItemMode( QwtLegendData::Checkable
            ↪ );
        insertLegend( legend, QwtPlot::RightLegend );
        connect( legend, SIGNAL( checked( const QVariant &,
            ↪ bool, int ) ),
```

```
            SLOT( showItem( const QVariant &, bool ) ) );
        replot(); // creating the legend items

        QwtPlotItemList items = itemList( QwtPlotItem::
            ↪ Rtti_PlotHistogram );
        for ( int i = 0; i < items.size(); i++ )
        {
            if ( i == 0 )
            {
                const QVariant itemInfo = itemToInfo( items[
                    ↪ i] );

                QwtLegendLabel *legendLabel =
                    qobject_cast<QwtLegendLabel *>( legend->
                        ↪ legendWidget( itemInfo ) );
                if ( legendLabel )
                    legendLabel->setChecked( true );

                items[i]->setVisible( true );
            }
            else
            {
                items[i]->setVisible( false );
            }
        }
    }else
        insertLegend( NULL );

}

void GraphPlot::showItem( const QVariant &itemInfo, bool on
    ↪ )
{
    QwtPlotItem *plotItem = infoToItem( itemInfo );
    if ( plotItem )
        plotItem->setVisible( on );
}
```

Listagem C.33: graph_plot.h

```
#ifndef GRAPHPLOT_H
#define GRAPHPLOT_H

#include <QtWidgets>
#include <qwt_plot.h>

#include "histogram_plot.h"
#include "mass_function_plot.h"
#include "density_function_plot.h"

class GraphPlot: public QwtPlot
{
```

```
    Q_OBJECT

public:
    GraphPlot(QWidget * = 0);
    void init();
    void plotHistogram(HistogramPlot *histogram);
    void enableLegends(bool enable);

private:
    int height_canvas_;

public slots:
    void plotProbalityFunctions(QList<QwtPlotCurve*> curves)
        ↪ ;
    void fitCanvasToHistogram(bool fit);

private slots:
    void showItem( const QVariant &, bool on );
};

#endif // GRAPHPLOT_H
```

Listagem C.34: graph_plot_factory.cpp

```
#include "utils/graph_plot_factory.h"

GraphPlotFactory::GraphPlotFactory()
{

}

DensityFunctionPlot *GraphPlotFactory::
    ↪ createDensityFunctionPlot(const QString name, const
    ↪ QColor color, const QList<QPointF> points)
{
    DensityFunctionPlot *density = new DensityFunctionPlot(
        ↪ name, color);
    density->setValues(points);
    return density;
}

MassFunctionPlot *GraphPlotFactory::createMassFunctionPlot(
    ↪ const QString name, const QColor color, const QList<
    ↪ QPointF> points)
{
    MassFunctionPlot *mass = new MassFunctionPlot(name,
        ↪ color);
    mass->setValues(points);
    return mass;
}
```

```
HistogramPlot *GraphPlotFactory::createHistogramPlot(QList<
    ↪ HistogramBin> bins, double highest_freq)
{
    HistogramPlot *hist_plot = new HistogramPlot("Histogram"
        ↪ , Qt::lightGray);
    hist_plot->setValues(bins);
    hist_plot->setHighestFrequency(highest_freq);
    return hist_plot;
}
```

Listagem C.35: graph_plot_factory.h

```
#ifndef GRAPHPLOTFACTORY_H
#define GRAPHPLOTFACTORY_H

#include "view/plot/density_function_plot.h"
#include "view/plot/histogram_plot.h"
#include "view/plot/mass_function_plot.h"

class GraphPlotFactory
{
public:
    GraphPlotFactory();
    static DensityFunctionPlot* createDensityFunctionPlot(
        ↪ const QString name, const QColor color,
                                                          const
                                                              ↪
                                                              ↪ QList
                                                              ↪ <
                                                              ↪ QPointF
                                                              ↪ >
                                                              ↪
                                                              ↪ points
                                                              ↪ )
                                                              ↪ ;
                                                              ↪
    static MassFunctionPlot* createMassFunctionPlot(const
        ↪ QString name, const QColor color,
                                                        const
                                                            ↪ QList
                                                            ↪ <
                                                            ↪ QPointF
                                                            ↪ >
                                                            ↪
                                                            ↪ points
                                                            ↪ )
                                                            ↪ ;
    static HistogramPlot* createHistogramPlot(QList<
        ↪ HistogramBin> bins, double highest_freq);
};

#endif // GRAPHPLOTFACTORY_H
```

Listagem C.36: histogram.cpp

```
#include "model/histogram.h"

#include <cmath>
#include <list>

#include "utils/utils.h"
#include "common/fit_utils.hpp"

Histogram::Histogram(const QList<double> samples, const fit
    ↪ ::_bin_criteria criteria,
                     const unsigned int max_bins, const int
                         ↪ num_bins):
    samples_(samples)
{
    highest_frequency_ = 0;

    if(Utils::checkDiscreteSamples(samples))
        type_ = fit::sample_type::DISCRETE;
    else
        type_ = fit::sample_type::CONTINUOUS;

    generateFrequencies(criteria, max_bins, num_bins);
}

Histogram::Histogram()
{

}

double Histogram::frequencyAt(double x)
{
    for(int i = 0; i < bins_.size() - 1; i++) {
        HistogramBin bin = bins_.at(i);
        if(x >= bin.getMin() && x < bin.getMax())
            return bin.getFrequency();
    }
    if(x >= bins_.last().getMin() && x <= bins_.last().
        ↪ getMax())
        return bins_.last().getFrequency();
    return -1;
}

void Histogram::generateFrequencies(const fit::_bin_criteria
    ↪ criteria,
                                    const unsigned int
                                        ↪ max_bins, const
                                        ↪ int num_bins)
{
```

```
    // sort
    std::list<double> sorted = samples_.toStdList();
    sorted.sort();
    samples_ = QList<double>::fromStdList(sorted);

    // get min and max
    double min = samples_.front();
    double max = samples_.back();
    double variation = max - min;

    // get number of bins
    double k;
    switch (criteria) {
    case fit::_bin_criteria::SQUAREROOT:
        k = std::round(std::sqrt((double)samples_.size()));
        break;
    case fit::_bin_criteria::STURGES:
        k = std::round (1.5 + 3.222 * std::log10((double)
            ↪ samples_.size()));
        break;
    case fit::_bin_criteria::OTHER:
        k = num_bins;
        break;
    default:
        k = max_bins;
    }

    k = k < max_bins ? k : max_bins;

    double width = variation / k;

    // round width up if discrete samples
    if(type_ == fit::sample_type::DISCRETE)
        width = std::ceil(width);

    width = width <= 0 ? 1 : width;

    // create bins
    createBins(min, width, (int)k);

    // update bin's frequency
    updateBinsFrequency();
}

void Histogram::createBins(double min, double width, const
    ↪ int k)
{
    bins_.clear();

    for(int i = 0; i < k; i++)
    {
        double max = min + width;
```

```
            HistogramBin bin(min, max, 0);
            bins_.push_back(bin);
            min = max;
        }
    }

    void Histogram::updateBinsFrequency()
    {
        QList<HistogramBin>::iterator current_bin = bins_.begin
            ↪ ();
        int i_bin = 0;
        for(int i = 0; i < samples_.size(); i++)
        {
            double sample = samples_.at(i);
            while(current_bin != bins_.end())
            {
                bool in_bound = sample >= current_bin->getMin()
                    ↪ && sample < current_bin->getMax();
                bool at_hist_right_bound = i_bin == bins_.size()
                    ↪ -1;

                if(in_bound || at_hist_right_bound)
                {
                    current_bin->incrementFrequency();
                    if(current_bin->getFrequency() >
                        ↪ highest_frequency_)
                        highest_frequency_ = current_bin->
                            ↪ getFrequency();
                    break;
                }
                else{
                    current_bin++;
                    i_bin++;
                }
            }
        }
    }

    double Histogram::getFrequencyAtMean()
    {
        return frequencyAt(fit::utils::_mean(samples_.toStdList
            ↪ ()));
    }

    double Histogram::getBinWidth()
    {
        if(bins_.isEmpty() == false)
            return bins_[0].binWidth();
        return 0;
    }

    int Histogram::getTotalOfSamples()
```

```
{
    return samples_.size();
}

void Histogram::setHighestFrequency(double highest_frequency
    ↪ )
{
    highest_frequency_ = highest_frequency;
}

// #### GETTERS ####
QList<HistogramBin> Histogram::getBins()
{
    return bins_;
}

double Histogram::getHighestFrequency() const
{
    return highest_frequency_;
}
```

Listagem C.37: histogram.h

```
#ifndef HISTOGRAM_H
#define HISTOGRAM_H

#include <QList>
#include "fit_defs.h"
#include "common/freq_table_entry.h"

#include "histogram_bin.h"
#include "defs.h"

class Histogram
{
public:
    Histogram(const QList<double> samples, const fit::
        ↪ _bin_criteria criteria,
            const unsigned int max_bins = fit::_MAX_BINS,
                ↪ const int num_bins = 0);
    Histogram();

    QList<HistogramBin> getBins();

    double getHighestFrequency() const;
    void setHighestFrequency(double highest_frequency);
    double getFrequencyAtMean();
    double getBinWidth();
    int getTotalOfSamples();

private:
    fit::sample_type type_;
```

```
    QList<double> samples_;
    QList<HistogramBin> bins_;
    double highest_frequency_;

    void generateFrequencies(const fit::_bin_criteria
        ↪ criteria, const unsigned int max_bins,
                                const int num_bins = 0);
    void createBins(double min, double width, const int k);
    void updateBinsFrequency();
    double frequencyAt(double x);
};

#endif // HISTOGRAM_H
```

Listagem C.38: histogram_bin.cpp

```
#include "model/histogram_bin.h"

HistogramBin::HistogramBin(const double min, const double
    ↪ max, const int frequency):
    min_(min), max_(max), frequency_(frequency)
{

}

void HistogramBin::incrementFrequency()
{
    frequency_++;
}

double HistogramBin::binWidth()
{
    return max_ - min_;
}

// #### GETTERS ####

double HistogramBin::getMin() const
{
    return min_;
}

double HistogramBin::getMax() const
{
    return max_;
}

int HistogramBin::getFrequency() const
{
    return frequency_;
}
```

```
// #### SETTERS ####

void HistogramBin::setMin(double min)
{
    min_ = min;
}

void HistogramBin::setFrequency(int frequency)
{
    frequency_ = frequency;
}

void HistogramBin::setMax(double max)
{
    max_ = max;
}
```

Listagem C.39: histogram_bin.h

```
#ifndef HISTOGRAMBIN_H
#define HISTOGRAMBIN_H


class HistogramBin
{
public:
    HistogramBin(const double min, const double max, const
        ↪ int frequency);

    void incrementFrequency();
    double binWidth();

    // SETTERS
    void setMin(double min);
    void setMax(double max);
    void setFrequency(int frequency);

    // GETTERS
    double getMin() const;
    double getMax() const;
    int getFrequency() const;

private:
    double min_;
    double max_;
    int frequency_;
};

#endif // HISTOGRAMBIN_H
```

Listagem C.40: histogram_plot.cpp

```
#include "view/plot/histogram_plot.h"

HistogramPlot::HistogramPlot(const QString &title, const
    ↪ QColor &color):
    QwtPlotHistogram(title)
{
    setStyle(QwtPlotHistogram::Columns);
    setColor(color);
}

void HistogramPlot::setColor(const QColor &color)
{
    QColor c = color;
    setBrush(QBrush(c));
}

void HistogramPlot::setValues(QList<HistogramBin> bins)
{
    QVector<QwtIntervalSample> samples(bins.size());
    for (int i = 0; i < bins.size(); i++)
    {
        QwtInterval interval(bins.at(i).getMin(), bins.at(i)
            ↪ .getMax());
        interval.setBorderFlags(QwtInterval::ExcludeMaximum
            ↪ );

        samples[i] = QwtIntervalSample(bins[i].getFrequency
            ↪ (), interval);
    }

    setData(new QwtIntervalSeriesData(samples));

    // set min and max of histogram
    min_ = bins.first().getMin();
    max_ = bins.last().getMax();
}
// #### GETTERS ####

double HistogramPlot::getMin()
{
    return min_;
}

double HistogramPlot::getMax()
{
    return max_;
}

double HistogramPlot::getHighestFrequency() const
{
    return highest_frequency_;
```

```
}

// #### SETTERS ####

void HistogramPlot::setHighestFrequency(double
    ↪ highest_frequency)
{
    highest_frequency_ = highest_frequency;
}

void HistogramPlot::setMin(double min)
{
    min_ = min;
}

void HistogramPlot::setMax(double max)
{
    max_ = max;
}
```

Listagem C.41: histogram_plot.h

```
#ifndef HISTOGRAMPLOT_H
#define HISTOGRAMPLOT_H

#include <qwt_plot_histogram.h>

#include "model/histogram_bin.h"

class HistogramPlot: public QwtPlotHistogram
{
public:
    HistogramPlot(const QString&, const QColor&);

    void setColor(const QColor&);
    void setValues(QList<HistogramBin> bins);

    // GETTERS
    double getMin();
    double getMax();
    double getHighestFrequency() const;

    // SETTERS
    void setHighestFrequency(double highest_frequency);
    void setMin(double min);
    void setMax(double max);

private:
    double min_, max_, highest_frequency_;
};

#endif // HISTOGRAMPLOT_H
```

Listagem C.42: lognorm_distrib.cpp

```
//
    ↪ ------------------------------------------------------------------------
    ↪
//     'FIT', a library for fitting statistical distribution
//     Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//     This library is free software; you can redistribute it
    ↪  and/or
//     modify it under the terms of the GNU Lesser General
    ↪ Public
//     License as published by the Free Software Foundation;
    ↪ either
//     version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//     This library is distributed in the hope that it will
    ↪ be useful,
//     but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//     Lesser General Public License for more details.
//
//     You should have received a copy of the GNU Lesser
    ↪ General Public
//     License along with this library; if not, write to the
    ↪ Free Software
//     Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪

#include <distribution/lognorm_distrib.h>
#include <cmath>
#include <algorithm>
#include <limits>
#include <common/fit_utils.hpp>

using namespace fit;
using std::exp;
using std::log;
using std::sqrt;

data_set lognorm_distrib::get_log_values(const data_set&
    ↪ values)
{
```

```
    data_set log_values;

    for (data_set::const_iterator it = values.begin(); it !=
        ↪  values.end(); it++)
    {
        if(*it == 0)
            log_values.push_back( std::numeric_limits<
                ↪ double >::min() );
        else
            log_values.push_back( std::log(*it) );
    }
    return log_values;
}

base_distrib *lognorm_distrib::clone()
{
    return new lognorm_distrib(*this);
}

std::string lognorm_distrib::name() const
{
    return "Lognormal";
}

lognorm_distrib::lognorm_distrib(const data_set& values)
{
    _samples_type = sample_type::CONTINUOUS;

    if (!check_values(values))
        throw distribution_exception("Invalid value on
            ↪ lognormal distribution.");

    _parameters.resize(PARAMS_NUMBER);
    set_params_mle(get_log_values(values));

    // init normal rand generator
    data_type sigma = _parameters[STDDEV];
    data_type sigma_sqr = sigma * sigma;
    data_type mu = _parameters[MEAN];
    data_type mu_1 = exp((mu+sigma_sqr)/2);
    data_type sigma_sqr_1 = exp(2*mu+sigma_sqr) * (exp(
        ↪ sigma_sqr) - 1);

    params_list params_normal;
    params_normal.push_back(log((mu_1*mu_1)/(sqrt(mu_1*mu_1
        ↪ + sigma_sqr_1))));
    params_normal.push_back(log(1 + (sigma_sqr_1/(mu_1*mu_1)
        ↪ )));
    norm = new norm_distrib(params_normal);
}

lognorm_distrib::lognorm_distrib(const params_list& params)
```

```
    : norm_distrib(params)
{
    // init normal rand generator
    data_type sigma = _parameters[STDDEV];
    data_type sigma_sqr = sigma * sigma;
    data_type mu = _parameters[MEAN];
    data_type mu_1 = exp((mu+sigma_sqr)/2);
    data_type sigma_sqr_1 = exp(2*mu+sigma_sqr) * (exp(
        ↪ sigma_sqr) - 1);

    params_list params_normal;
    params_normal.push_back(log((mu_1*mu_1)/(sqrt(mu_1*mu_1
        ↪ + sigma_sqr_1))));
    params_normal.push_back(log(1 + (sigma_sqr_1/(mu_1*mu_1)
        ↪ )));
    norm = new norm_distrib(params_normal);
}

lognorm_distrib::~lognorm_distrib()
{
    delete norm;
}

data_type lognorm_distrib::get_mean() const
{
    data_type sigma = _parameters[STDDEV];
    return exp(_parameters[MEAN] + ((sigma * sigma) / 2));
}

data_type lognorm_distrib::get_variance() const
{
    data_type sigma = _parameters[STDDEV];
    data_type sigma_sqr = sigma * sigma;
    return exp((2 * _parameters[MEAN]) + sigma_sqr) * (exp(
        ↪ sigma_sqr) - 1);
}

data_type lognorm_distrib::get_mode() const
{
    data_type sigma = _parameters[STDDEV];
    data_type sigma_sqr = sigma * sigma;
    return exp(_parameters[MEAN] - sigma_sqr);
}

data_type lognorm_distrib::cumulative_function(data_type
    ↪ number) const
{
    return norm_distrib::cumulative_function( std::log(
        ↪ number) );
}

data_type lognorm_distrib::probability_function(data_type
```

```
    ↪ number) const
{
    if (number <= 0)
        return 0;

    data_type sigma = _parameters[STDDEV];
    data_type sigma_sqr = sigma * sigma;
    data_type mu = _parameters[MEAN];

    data_type numer = exp( (-pow(log(number) - mu, 2)) / (2
        ↪ * sigma_sqr) );
    data_type denom = number * sigma * sqrt(2 * utils::PI);

    return numer / denom;
}

data_type lognorm_distrib::get_random() const
{
    return exp(norm->get_random());
}
```

Listagem C.43: lognorm_distrib.h

```
//
    ↪ ------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
```

```
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------
    ↪

#ifndef LOGNORM_DISTRIB_H_
#define LOGNORM_DISTRIB_H_

/*
 * Lognormal Distribution
 */

#include "norm_distrib.h"

namespace fit
{
class lognorm_distrib : public norm_distrib
{
private:
    norm_distrib *norm;
    static data_set get_log_values(const data_set& values);

protected:
    inline bool in_range(data_type value) const { return
        ↪ value >= 0; }

public:
    lognorm_distrib(const data_set& values); //gets
        ↪ distribution parameters from MLE
    lognorm_distrib(const params_list& params); //gets
        ↪ distribution parameters from params.

    base_distrib *clone();

    std::string name() const;

    ~lognorm_distrib();

    data_type get_mean() const;
    data_type get_variance() const;
    data_type get_mode() const;

    data_type cumulative_function(data_type number) const;
    data_type probability_function(data_type number) const;
        ↪ //probability density function

    /* Gets a random value according to distribution */
    data_type get_random() const;
};
}
```

```
#endif /*LOGNORM_DISTRIB_H_*/
```

Listagem C.44: main.cpp

```
#include "view/main_window.h"

#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    a.setApplicationName("GOFTester");
    a.setOrganizationName("UFSC");
    a.setOrganizationDomain("ufsc.br");
    MainWindow w;
    w.show();

    return a.exec();
}
```

Listagem C.45: main_window.cpp

```
#include "view/main_window.h"
#include "ui_main_window.h"

#include <QSettings>
#include <QMessageBox>

#include "view/plot/histogram_plot.h"
#include "utils/file_handler.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui_(new Ui::MainWindow)
{
    ui_->setupUi(this);

    // init graph plot
    ui_->widget_graph_plot->init();

    setUpConnections();

    ui_->button_back->setHidden(true);
    ui_->tree_widget_distrib_info->hide();
    ui_->splitter_3->setStretchFactor(0, 0);
    ui_->splitter_3->setStretchFactor(1, 1);
}

MainWindow::~MainWindow()
{
    delete ui_;
```

```
}

void MainWindow::setSamples(QList<double> samples)
{
    samples_ = samples;
    gof_tester_.setSamples(samples);
}

void MainWindow::getRandSamples(const int numb_samples,
    ↪ const QVector<double> params, const int distrib_type)
{
    ui_->widget_screen_1->setSamples(gof_tester_.
        ↪ getRandomSamples(numb_samples, params,
        ↪ distrib_type));
}

void MainWindow::createHistogram(const int bin_criteria,
    ↪ const int num_bins)
{
    ui_->widget_graph_plot->detachItems(QwtPlotItem::
        ↪ Rtti_PlotHistogram, true);
    ui_->widget_graph_plot->plotHistogram(gof_tester_.
        ↪ createHistogram(bin_criteria, num_bins));
    showHistogramInfo();

    ui_->button_next->setEnabled(true);
}

void MainWindow::showFitResults(QList<DistributionFit*>
    ↪ distributions_fit, QStringList distrib_error_fit)
{
    showDistribInfo(distributions_fit);

    if(distrib_error_fit.isEmpty() == false)
    {
        QString list_distrib = "";
        for(int i = 0; i < distrib_error_fit.size(); i++)
        {
            if(i == distrib_error_fit.size()-1)
                list_distrib += distrib_error_fit[i];
            else
                list_distrib += distrib_error_fit[i] + ", ";
        }
        QMessageBox::warning(this, "Warning", "The following
            ↪  distributions could not be fitted:\n\n"
                                            + list_distrib
                                                ↪ );

    }
    ui_->widget_screen_3->showChiResults(distributions_fit);
}
```

```
void MainWindow::addRootToTreeWidget(DistributionFit*
    ↪ distribution)
{
    QTreeWidgetItem *item = new QTreeWidgetItem(ui_->
        ↪ tree_widget_distrib_info);
    item->setText(0, distribution->getName());
    ui_->tree_widget_distrib_info->addTopLevelItem(item);

    QVector<QPair<QString,double> > params = distribution->
        ↪ getParams();
    for(int i = 0; i < params.size(); i++)
        addChildToTreeWidget(item, params.at(i).first,
            ↪ QString::number(params.at(i).second));

    addChildToTreeWidget(item, "offset", QString::number(
        ↪ distribution->getOffset()));
    addChildToTreeWidget(item, "scale factor", QString::
        ↪ number(distribution->getScaleFactor()));

    item->setExpanded(true);
}

void MainWindow::addChildToTreeWidget(QTreeWidgetItem *
    ↪ parent, const QString param, const QString value)
{
    QTreeWidgetItem *item = new QTreeWidgetItem();
    item->setText(0, param);
    item->setText(1, value);
    parent->addChild(item);
}

void MainWindow::showHistogramInfo()
{
    QList<QPair<QString, QPair<int, double> > > info =
        ↪ gof_tester_.getHistogramInfo();

    ui_->table_frequency->setRowCount(info.size());
    for(int i = 0; i < info.size(); i++)
    {
        QString range = info[i].first;
        QPair<int, double> frequency = info[i].second;

        ui_->table_frequency->setVerticalHeaderItem(i, new
            ↪ QTableWidgetItem(range));
        ui_->table_frequency->setItem(i, 0, new
            ↪ QTableWidgetItem(QString::number(frequency.
            ↪ first)));
        ui_->table_frequency->setItem(i, 1, new
            ↪ QTableWidgetItem(QString::number(frequency.
            ↪ second, 'g', 2) + "%"));
    }
}
```

```
void MainWindow::showDistribInfo(QList<DistributionFit *>
    ↪ distributions_fit)
{
    foreach(DistributionFit *distrib, distributions_fit)
    {
        addRootToTreeWidget(distrib);
    }

    ui_->tree_widget_distrib_info->resizeColumnToContents(0)
        ↪ ;
}

void MainWindow::on_button_next_clicked()
{
    QSettings settings;
    if(ui_->stacked_widget->currentIndex() == 0)
    {
        ui_->stacked_widget->setCurrentIndex(1);
        ui_->button_back->show();
    }
    else if(ui_->stacked_widget->currentIndex() == 1)
    {
        QList<int> indexes = ui_->widget_screen_2->
            ↪ getSelectedDistributions();
        if(indexes.isEmpty())
        {
            QMessageBox::critical(this, "Error", "Select at
                ↪ least one distribution.");
            return;
        }
        ui_->stacked_widget->setCurrentIndex(2);
        int bin_criteria = ui_->widget_screen_2->
            ↪ getBinCriteria();
        ui_->button_next->setDisabled(true);
        ui_->widget_graph_plot->enableLegends(true);
        ui_->tree_widget_distrib_info->show();
        gof_tester_.fitDistributions(indexes, bin_criteria);
    }
    ui_->splitter_2->restoreState(settings.value("
        ↪ splitterSizes").toByteArray());
}

void MainWindow::on_button_back_clicked()
{
    QSettings settings;
    if(ui_->stacked_widget->currentIndex() == 1)
    {
        // page 2
        ui_->button_back->hide();
        ui_->stacked_widget->setCurrentIndex(0);
    }else
```

```
    {
        // page 3
        ui_->button_next->setEnabled(true);
        ui_->widget_graph_plot->enableLegends(false);
        ui_->tree_widget_distrib_info->hide();
        ui_->stacked_widget->setCurrentIndex(1);

        // reset widgets
        ui_->widget_graph_plot->detachItems(QwtPlotItem::
            ↪ Rtti_PlotCurve);
        ui_->tree_widget_distrib_info->clear();
    }
    ui_->splitter_2->restoreState(settings.value("
        ↪ splitterSizes").toByteArray());
}

void MainWindow::exportGraph()
{
    const QString DEFAULT_DIR_KEY("default_dir");
    QSettings my_settings;

    QString filters = "PNG (*.png);;JPEG (*.jpg);;SVG (*.svg
        ↪ );;PDF (*.pdf)";
    QString default_filter = "JPEG (*.jpg)";
    QString file = QFileDialog::getSaveFileName(this,
        tr("Export graph as"), my_settings.value(
            ↪ DEFAULT_DIR_KEY).toString(), filters, &
            ↪ default_filter);
    if (!file.isEmpty())
    {
        // saving last visited dir
        QFile file_aux(file);
        QFileInfo file_info(file_aux.fileName());
        QString path(file_info.path());
        my_settings.setValue(DEFAULT_DIR_KEY, path);

        if(default_filter == "PNG (*.png)")
            file += ".png";
        if(default_filter == "JPEG (*.jpg)")
            file += ".jpg";
        if(default_filter == "SVG (*.svg)")
            file += ".svg";
        if(default_filter == "PDF (*.pdf)")
            file += ".pdf";

        FileHandler::exportGraph(ui_->widget_graph_plot,
            ↪ file);
    }
}

void MainWindow::saveSamplesToFile()
{
```

```
    if(samples_.isEmpty())
    {
        QMessageBox::critical(this, "Error", "No existing
            ↪ samples to save.");
        return;
    }

    const QString DEFAULT_DIR_KEY("default_dir");
    QSettings my_settings;

    QString selected_filter;
    QString file = QFileDialog::getSaveFileName(this, tr("
        ↪ Save samples as Excel file"),
                                                my_settings.
                                                    ↪ value
                                                    ↪ (
                                                    ↪ DEFAULT_DIR_KE
                                                    ↪ ).
                                                    ↪ toString
                                                    ↪ (),
                                                tr("Excel
                                                    ↪ file
                                                    ↪ (*.
                                                    ↪ xlsx)
                                                    ↪ ;;Txt
                                                    ↪  file
                                                    ↪  (*.
                                                    ↪ txt)"
                                                    ↪ ),
                                                &
                                                    ↪ selected_filte
                                                    ↪ );
    if (!file.isEmpty())
    {
        // saving last visited dir
        QFile file_aux(file);
        QFileInfo file_info(file_aux.fileName());
        QString path(file_info.path());
        my_settings.setValue(DEFAULT_DIR_KEY, path);

        if(selected_filter == "Excel file (*.xlsx)")
            FileHandler::writeSamplesToXlsx(samples_, file);
        else
            FileHandler::writeSamplesToTxt(samples_, file);
    }
}

void MainWindow::on_splitter_2_splitterMoved(int pos, int
    ↪ index)
{
    Q_UNUSED(index);
    if(pos > 0)
```

```
    {
        QSettings settings;
        settings.setValue("splitterSizes", ui->splitter_2->
            ↪ saveState());
    }
}

void MainWindow::showEvent(QShowEvent *event)
{
    Q_UNUSED(event);
    QSettings settings;

    settings.setValue("splitterSizes", ui->splitter_2->
        ↪ saveState());

    QList<int> sizes;
    sizes.append(150);
    sizes.append(200);
    ui->splitter_3->setSizes(sizes);
}

void MainWindow::setUpConnections()
{
    connect(ui->widget_screen_1, SIGNAL(sigGenerateRandNumb
        ↪ (int, QVector<double>, int)),
            this, SLOT(getRandSamples(int, QVector<double>,
                ↪ int)));
    connect(ui->widget_screen_1, SIGNAL(sigCreateHistogram(
        ↪ int,int)),
            this, SLOT(createHistogram(int,int)));
    connect(ui->widget_screen_1, SIGNAL(sigSetSamples(QList
        ↪ <double>)),
            this, SLOT(setSamples(QList<double>)));
    connect(ui->widget_screen_1, SIGNAL(sigEnableNext(bool)
        ↪ ), ui->button_next, SLOT(setEnabled(bool)));
    connect(&gof_tester_, SIGNAL(sigPlotDistributions(QList<
        ↪ QwtPlotCurve*>)),
            ui->widget_graph_plot, SLOT(
                ↪ plotProbalityFunctions(QList<QwtPlotCurve
                ↪ *>)));
    connect(&gof_tester_, SIGNAL(sigShowDistributionsInfo(
        ↪ QList<DistributionFit*>,QStringList)),
            this, SLOT(showFitResults(QList<DistributionFit
                ↪ *>, QStringList)));
    connect(ui->actionExport_graph, SIGNAL(triggered()),
        ↪ this, SLOT(exportGraph()));
    connect(ui->actionFix_plot_area_to_histogram, SIGNAL(
        ↪ triggered(bool)),
            ui->widget_graph_plot, SLOT(
                ↪ fitCanvasToHistogram(bool)));
    connect(ui->action_export_samples, SIGNAL(triggered()),
            this, SLOT(saveSamplesToFile()));
```

```
}
```

Listagem C.46: main_window.h

```
#ifndef MAIN_WINDOW_H
#define MAIN_WINDOW_H

#include <QMainWindow>
#include <QTreeWidgetItem>

#include "model/gof_tester.h"
#include "view/plot/density_function_plot.h"
#include "view/plot/mass_function_plot.h"

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

protected:
    virtual void showEvent(QShowEvent* event);

private:
    Ui::MainWindow *ui_;
    GOFTester gof_tester_;
    QList<double> samples_;

    void setUpConnections();
    void addRootToTreeWidget(DistributionFit* distribution);
    void addChildToTreeWidget(QTreeWidgetItem *parent, const
        ↪ QString param, const QString value);
    void showHistogramInfo();
    void showDistribInfo(QList<DistributionFit*>
        ↪ distributions_fit);

public slots:
    void setSamples(QList<double> samples);
    void getRandSamples(const int numb_samples, const
        ↪ QVector<double> params, const int distrib_type);
    void createHistogram(const int bin_criteria, const int
        ↪ num_bins = 0);
    void showFitResults(QList<DistributionFit*>
        ↪ distributions_fit, QStringList distrib_error_fit)
        ↪ ;
```

```
private slots:
    void on_button_next_clicked();
    void on_splitter_2_splitterMoved(int pos, int index);
    void on_button_back_clicked();
    void exportGraph();
    void saveSamplesToFile();

signals:
    void sigEnableNext(bool enabled);
};

#endif // MAIN_WINDOW_H
```

Listagem C.47: main_window.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>601</width>
    <height>468</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>GOFTester</string>
  </property>
  <widget class="QWidget" name="central_widget">
   <layout class="QVBoxLayout" name="verticalLayout" stretch
       ↪ ="1,0">
    <property name="topMargin">
     <number>0</number>
    </property>
    <property name="rightMargin">
     <number>0</number>
    </property>
    <item>
     <widget class="QSplitter" name="splitter">
      <property name="orientation">
       <enum>Qt::Horizontal</enum>
      </property>
      <widget class="QSplitter" name="splitter_2">
       <property name="orientation">
        <enum>Qt::Vertical</enum>
       </property>
       <widget class="QStackedWidget" name="stacked_widget">
        <property name="currentIndex">
         <number>0</number>
        </property>
```

```
<widget class="QWidget" name="page">
 <layout class="QVBoxLayout" name="verticalLayout_2"
     ↪ >
  <property name="leftMargin">
   <number>0</number>
  </property>
  <property name="topMargin">
   <number>0</number>
  </property>
  <property name="rightMargin">
   <number>0</number>
  </property>
  <property name="bottomMargin">
   <number>0</number>
  </property>
  <item>
   <widget class="WidgetSamplesAndHistogram" name="
       ↪ widget_screen_1" native="true"/>
  </item>
 </layout>
</widget>
<widget class="QWidget" name="page_2">
 <layout class="QVBoxLayout" name="verticalLayout_3"
     ↪ >
  <property name="leftMargin">
   <number>0</number>
  </property>
  <property name="topMargin">
   <number>0</number>
  </property>
  <property name="rightMargin">
   <number>0</number>
  </property>
  <property name="bottomMargin">
   <number>0</number>
  </property>
  <item>
   <widget class="WidgetDistribToFit" name="
       ↪ widget_screen_2" native="true"/>
  </item>
 </layout>
</widget>
<widget class="QWidget" name="page_3">
 <layout class="QVBoxLayout" name="verticalLayout_6"
     ↪ >
  <property name="spacing">
   <number>0</number>
  </property>
  <property name="leftMargin">
   <number>0</number>
  </property>
  <property name="topMargin">
```

```
      <number>0</number>
     </property>
     <property name="rightMargin">
      <number>0</number>
     </property>
     <property name="bottomMargin">
      <number>0</number>
     </property>
     <item>
      <widget class="WidgetChiResults" name="
          ↪ widget_screen_3" native="true"/>
     </item>
    </layout>
   </widget>
  </widget>
  <widget class="QWidget" name="verticalLayoutWidget">
   <layout class="QVBoxLayout" name="vertical_layout">
    <property name="rightMargin">
     <number>6</number>
    </property>
    <item>
     <widget class="QFrame" name="frame_3">
      <property name="frameShape">
       <enum>QFrame::Box</enum>
      </property>
      <property name="frameShadow">
       <enum>QFrame::Raised</enum>
      </property>
      <property name="lineWidth">
       <number>2</number>
      </property>
      <layout class="QVBoxLayout" name="
          ↪ verticalLayout_5">
       <item>
        <widget class="QSplitter" name="splitter_3">
         <property name="orientation">
          <enum>Qt::Horizontal</enum>
         </property>
         <widget class="QTableWidget" name="
             ↪ table_frequency">
          <property name="sizePolicy">
           <sizepolicy hsizetype="Preferred" vsizetype=
               ↪ "Expanding">
            <horstretch>0</horstretch>
            <verstretch>0</verstretch>
           </sizepolicy>
          </property>
          <property name="editTriggers">
           <set>QAbstractItemView::NoEditTriggers</set>
          </property>
          <attribute name="
              ↪ horizontalHeaderStretchLastSection">
```

```
                <bool>true</bool>
               </attribute>
               <attribute name="
                   ↪ verticalHeaderStretchLastSection">
                <bool>false</bool>
               </attribute>
               <column>
                <property name="text">
                 <string>Frequency</string>
                </property>
               </column>
               <column>
                <property name="text">
                 <string>Percentage</string>
                </property>
               </column>
              </widget>
              <widget class="GraphPlot" name="
                  ↪ widget_graph_plot" native="true">
               <property name="sizePolicy">
                <sizepolicy hsizetype="Expanding" vsizetype=
                    ↪ "Expanding">
                 <horstretch>0</horstretch>
                 <verstretch>0</verstretch>
                </sizepolicy>
               </property>
              </widget>
             </widget>
            </item>
           </layout>
          </widget>
         </item>
        </layout>
       </widget>
      </widget>
      <widget class="QTreeWidget" name="
          ↪ tree_widget_distrib_info">
       <property name="baseSize">
        <size>
         <width>0</width>
         <height>0</height>
        </size>
       </property>
       <property name="styleSheet">
        <string notr="true"/>
       </property>
       <property name="alternatingRowColors">
        <bool>true</bool>
       </property>
       <property name="selectionMode">
        <enum>QAbstractItemView::NoSelection</enum>
       </property>
```

```
    <attribute name="headerCascadingSectionResizes">
     <bool>true</bool>
    </attribute>
    <column>
     <property name="text">
      <string>Distribution</string>
     </property>
    </column>
    <column>
     <property name="text">
      <string>Value</string>
     </property>
    </column>
   </widget>
  </widget>
 </item>
 <item>
  <layout class="QHBoxLayout" name="horizontal_layout">
   <item>
    <spacer name="horizontal_spacer">
     <property name="orientation">
      <enum>Qt::Horizontal</enum>
     </property>
     <property name="sizeHint" stdset="0">
      <size>
       <width>40</width>
       <height>20</height>
      </size>
     </property>
    </spacer>
   </item>
   <item>
    <widget class="QPushButton" name="button_back">
     <property name="enabled">
      <bool>true</bool>
     </property>
     <property name="text">
      <string>&lt; Back</string>
     </property>
    </widget>
   </item>
   <item>
    <widget class="QPushButton" name="button_next">
     <property name="enabled">
      <bool>false</bool>
     </property>
     <property name="text">
      <string>Next &gt;</string>
     </property>
    </widget>
   </item>
   <item>
```

```
      <spacer name="horizontal_spacer_2">
       <property name="orientation">
        <enum>Qt::Horizontal</enum>
       </property>
       <property name="sizeType">
        <enum>QSizePolicy::Fixed</enum>
       </property>
       <property name="sizeHint" stdset="0">
        <size>
         <width>20</width>
         <height>20</height>
        </size>
       </property>
      </spacer>
     </item>
    </layout>
   </item>
  </layout>
 </widget>
 <widget class="QMenuBar" name="menuBar">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>601</width>
    <height>17</height>
   </rect>
  </property>
  <widget class="QMenu" name="menuFile">
   <property name="title">
    <string>Samples</string>
   </property>
   <addaction name="action_export_samples"/>
  </widget>
  <widget class="QMenu" name="menuGraph">
   <property name="title">
    <string>Graph</string>
   </property>
   <addaction name="actionFix_plot_area_to_histogram"/>
   <addaction name="actionExport_graph"/>
  </widget>
  <addaction name="menuFile"/>
  <addaction name="menuGraph"/>
 </widget>
 <widget class="QStatusBar" name="statusBar"/>
 <action name="actionExport">
  <property name="text">
   <string>Export</string>
  </property>
 </action>
 <action name="action_export_samples">
  <property name="text">
```

```
    <string>Export</string>
   </property>
  </action>
  <action name="actionSamples_as_xlsx">
   <property name="text">
    <string>Samples as xlsx</string>
   </property>
  </action>
  <action name="actionGraph">
   <property name="text">
    <string>Graph</string>
   </property>
  </action>
  <action name="actionFix_plot_area_to_histogram">
   <property name="checkable">
    <bool>true</bool>
   </property>
   <property name="text">
    <string>Fix plot area to histogram</string>
   </property>
  </action>
  <action name="actionExport_graph">
   <property name="text">
    <string>Export graph</string>
   </property>
  </action>
 </widget>
 <layoutdefault spacing="6" margin="11"/>
 <customwidgets>
  <customwidget>
   <class>GraphPlot</class>
   <extends>QWidget</extends>
   <header>view/plot/graph_plot.h</header>
   <container>1</container>
  </customwidget>
  <customwidget>
   <class>WidgetSamplesAndHistogram</class>
   <extends>QWidget</extends>
   <header>view/widget_samples_and_histogram.h</header>
   <container>1</container>
  </customwidget>
  <customwidget>
   <class>WidgetDistribToFit</class>
   <extends>QWidget</extends>
   <header>view/widget_distrib_to_fit.h</header>
   <container>1</container>
  </customwidget>
  <customwidget>
   <class>WidgetChiResults</class>
   <extends>QWidget</extends>
   <header>view/widget_chi_results.h</header>
   <container>1</container>
```

```
  </customwidget>
 </customwidgets>
 <resources/>
 <connections/>
</ui>
```

Listagem C.48: mass_function_plot.cpp

```
#include "view/plot/mass_function_plot.h"

#include <qwt_symbol.h>

MassFunctionPlot::MassFunctionPlot(const QString title,
    ↪ const QColor& color)
{
    //setSymbol(new QwtSymbol(QwtSymbol::Ellipse, Qt::yellow
        ↪ ,
    //                                QPen( Qt::blue), QSize(5,5))
        ↪ );
    setTitle(title);
    setPen(color, 2);
    setStyle(QwtPlotCurve::Steps);
}

void MassFunctionPlot::setValues(QList<QPointF> points)
{
    QPolygonF points_plot;
    foreach(QPointF point, points)
    {
        points_plot << point;
    }
    setSamples(points_plot);
}
```

Listagem C.49: mass_function_plot.h

```
#ifndef MASSFUNCTIONPLOT_H
#define MASSFUNCTIONPLOT_H

#include "qwt_plot_curve.h"

class MassFunctionPlot: public QwtPlotCurve
{
public:
    MassFunctionPlot(const QString title, const QColor&
        ↪ color);
    void setValues(QList<QPointF> points);
};

#endif // MASSFUNCTIONPLOT_H
```

Listagem C.50: norm_distrib.cpp

```
//
    ↪ ------------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪

#include <distribution/norm_distrib.h>
#include <cmath>
#include <common/fit_utils.hpp>

using namespace fit;
using std::sqrt;
using std::log;
using std::sin;
using std::abs;
using std::exp;

norm_distrib::norm_distrib(const data_set& values)
{
    _samples_type = sample_type::CONTINUOUS;

    _parameters.resize(PARAMS_NUMBER);
```

```
    set_params_mle(values);
}

base_distrib *norm_distrib::clone()
{
    return new norm_distrib(*this);
}

std::string norm_distrib::name() const
{
    return "Normal";
}

std::vector<std::string> norm_distrib::paramNames() const
{
    std::vector<std::string> paramNames = std::vector<std::
        ↪ string>();
    paramNames.push_back("Mean");
    paramNames.push_back("Standard deviation");

    return paramNames;
}

norm_distrib::norm_distrib(const params_list& params)
{
    _samples_type = sample_type::CONTINUOUS;

    if (!check_parameters(params))
        throw distribution_exception("Invalid parameters on
            ↪ normal distribution.");

    _parameters = params;
}

data_type norm_distrib::cumulative_function(data_type number
    ↪ ) const
{
    data_type z_number = ( number - _parameters[MEAN] ) /
        ↪ _parameters[STDDEV];
    return normal_cdf(z_number);
}

data_type norm_distrib::probability_function(data_type
    ↪ number) const
{
    data_type mu = _parameters[MEAN];
    data_type sigsqr = get_variance();
    data_type numer = exp( -((number - mu) * (number - mu))
        ↪ / (2 * sigsqr) );
    data_type denom = sqrt(2 * sigsqr * utils::PI);

    return numer / denom;
```

```
}

void norm_distrib::set_params_mle(const data_set& values)
{
    data_type md = utils::_mean(values);
    data_type sigma = sqrt(utils::_variance(md, values));

    _parameters[MEAN] = md;
    _parameters[STDDEV] = sigma;
}

data_type norm_distrib::get_mean() const
{
    return _parameters[MEAN];
}

data_type norm_distrib::get_variance() const
{
    return std::pow(_parameters[STDDEV], 2);
}

data_type norm_distrib::get_mode() const
{
    return _parameters[MEAN];
}

data_type norm_distrib::get_random() const
{
    data_type z = sqrt( -2 * log(random_number()) ) * sin(
        ↪ 6.2831853071 * random_number() );
    return _parameters[MEAN] + (z * _parameters[STDDEV]);
}

bool norm_distrib::check_parameters(const params_list&
    ↪ params) const
{
    return ((params.size() == PARAMS_NUMBER) && (params.at(
        ↪ STDDEV) > 0));
}

//-------------------------------------------------
int norm_distrib::sgn(const data_type& x)
{
    return x < 0.0 ? -1 : 1;
}

data_type norm_distrib::g(data_type x, data_type u)
{
    return 0.5 + u * sgn(x);
}

data_type norm_distrib::normal_cdf(data_type x)
```

```
{
    const data_type ONE_SQRT_2_PI = 0.39894228040143267793;

    double y = abs(x);

    if (y > 7.335)
        return g(x);

    if (y < 0.0001) {
        long double u = y * ONE_SQRT_2_PI;
        return g(x, u);
    }

    long double a = 0.0;
    long double b = 1.0;
    long double u = 1.0;
    long double j = -1.0;
    int n = int(4 + 4 * y);
    long double z = 1.0 / (y * y);
    long double v = 1.0 / (4 * n + 2);
    long double d = 0;

    for (int i = n; i > 0; --i) {
        d = a - 8 * i * b * z;
        u = u + d * j;
        v = v + (d - b) / (4 * i - 2);
        a = b;
        b = d;
        j = -j;
    }

    u = (-v * y * ONE_SQRT_2_PI) / (u * j + d / 2.0);
    return g(x, u);
}
```

Listagem C.51: norm_distrib.h

```
//
    ↪ ------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
```

```
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪

#ifndef NORM_DISTRIB_H_
#define NORM_DISTRIB_H_

/*
 * Normal Distribution
 */

#include "base_distrib.h"

namespace fit
{
class norm_distrib : public base_distrib
{
protected:
    static inline int sgn(const data_type& x);
    static inline data_type g(data_type x, data_type u =
        ↪ 0.5);
    static data_type normal_cdf(data_type x); //normal's
        ↪ cumulative distribution function.

    norm_distrib() {};

public:
    enum _norm_params
    {
        MEAN,
        STDDEV,
        PARAMS_NUMBER
    };
    norm_distrib(const data_set& values); //gets
        ↪ distribution parameters from MLE
    norm_distrib(const params_list& params); //gets
        ↪ distribution parameters from params.
```

```
    //Return a clone of de distribuition allocated on heap
    base_distrib *clone();

    std::string name() const;
    virtual std::vector<std::string> paramNames() const;

    virtual ~norm_distrib() {}

    virtual data_type get_mean() const;
    virtual data_type get_variance() const;
    virtual data_type get_mode() const;

    virtual data_type cumulative_function(data_type number)
        ↪ const;
    virtual data_type probability_function(data_type number)
        ↪  const; //probability density function

    /* Sets distribution parameters through maximum
        ↪ likelihood procedures */
    void set_params_mle(const data_set& values);

    /* Gets a random value according to distribution */
    virtual data_type get_random() const;

    bool check_parameters(const params_list& params) const;
};
}

#endif /*NORM_DISTRIB_H_*/
```

Listagem C.52: poisson_distrib.cpp

```
#include "distribution/poisson_distrib.h"

#include "common/fit_utils.hpp"

#include <cmath>

using namespace fit;

poisson_distrib::poisson_distrib(const data_set& values)
{
    if(utils::_check_samples_type(values) != sample_type::
        ↪ DISCRETE)
        throw distribution_exception("Poisson samples must
            ↪ be integer.");
    _samples_type = sample_type::DISCRETE;
    _parameters.resize(PARAMS_NUMBER);
    set_params_mle(values);
}
```

```
base_distrib *poisson_distrib::clone()
{
    return new poisson_distrib(*this);
}

std::string poisson_distrib::name() const
{
    return "Poisson";
}

std::vector<std::string> poisson_distrib::paramNames() const
{
    std::vector<std::string> paramNames = std::vector<std::
        ↪ string>();
    paramNames.push_back("Lambda");

    return paramNames;
}

poisson_distrib::poisson_distrib(const params_list& params)
{
    _samples_type = sample_type::DISCRETE;
    if (!check_parameters(params))
        throw distribution_exception("Invalid parameters on
            ↪ poisson distribution.");

    _parameters = params;
}

data_type poisson_distrib::cumulative_function(data_type
    ↪ number) const
{
    if(number < 1)
        return exp(-number);

    return utils::gammq(number, _parameters[LAMBDA]);
}

data_type poisson_distrib::probability_function(data_type
    ↪ number) const
{
    data_type lambda= _parameters[LAMBDA];
    return exp(number * log(lambda) - lambda - utils::
        ↪ gammln(number));
}

void poisson_distrib::set_params_mle(const data_set& values)
{
    _parameters[LAMBDA] = utils::_mean(values);
}

data_type poisson_distrib::get_mean() const
```

```
{
    return _parameters[LAMBDA];
}

data_type poisson_distrib::get_variance() const
{
    return _parameters[LAMBDA];
}

data_type poisson_distrib::get_mode() const
{

    return floor(_parameters[LAMBDA]); // or return lambda -
        ↪ 1;
}

data_type poisson_distrib::get_random() const
{
    data_type n = 0, p = 1;
    data_type lambda = _parameters[LAMBDA];

    while(true)
    {
        data_type r = random_number();
        p = p*r;

        if(p < exp(-lambda))
            return n;
        else
            n++;
    }
}

bool poisson_distrib::check_parameters(const params_list&
    ↪ params) const
{
    return ((params.size() == PARAMS_NUMBER) && (params.at(
        ↪ LAMBDA) > 0));
}
```

Listagem C.53: poisson_distrib.h

```
#ifndef POISSON_DISTRIB_H
#define POISSON_DISTRIB_H

/*
 * Poisson Distribution
 */

#include "base_distrib.h"

namespace fit
```

```
{
class poisson_distrib: public base_distrib
{
protected:
    inline bool in_range(data_type value) const {return
        ↪ value >= 0; }

public:
    enum _poisson_params
    {
        LAMBDA,
        PARAMS_NUMBER
    };
    poisson_distrib(const data_set& values); //gets
        ↪ distribution parameters from MLE
    poisson_distrib(const params_list& params); //gets
        ↪ distribution parameters from params.

    //Return a clone of de distribuition allocated on heap
    base_distrib *clone();

    std::string name() const;
    std::vector<std::string> paramNames() const;

    virtual ~poisson_distrib() {}

    virtual data_type get_mean() const;
    virtual data_type get_variance() const;
    virtual data_type get_mode() const;

    virtual data_type cumulative_function(data_type number)
        ↪ const;
    virtual data_type probability_function(data_type number)
        ↪  const; //probability mass function

    /* Sets distribution parameters through maximum
        ↪ likelihood procedures */
    void set_params_mle(const data_set& values);

    /* Gets a random value according to distribution */
    virtual data_type get_random() const;

    bool check_parameters(const params_list& params) const;
};
}

#endif // POISSON_DISTRIB_H
```

Listagem C.54: rsc.qrc

```
<RCC>
    <qresource prefix="/selectable_distrib">
```

```
        <file>uniform.svg</file>
        <file>weibull.svg</file>
        <file>beta_graph.svg</file>
        <file>disc_unif_graph.svg</file>
        <file>expo_graph.svg</file>
        <file>gamma_graph.svg</file>
        <file>lognormal.svg</file>
        <file>normal_graph.svg</file>
        <file>poisson_graph.svg</file>
        <file>triang_graph.svg</file>
    </qresource>
</RCC>
```

Listagem C.55: tria_distrib.cpp

```
//
    ↪ ------------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪

#include <distribution/tria_distrib.h>
#include <algorithm>
#include <cmath>
```

```
#include <common/fit_utils.hpp>

using namespace fit;
using std::sqrt;

tria_distrib::tria_distrib(const data_set& values)
{
    _samples_type = sample_type::CONTINUOUS;

    _parameters.resize(PARAMS_NUMBER);
    set_params_mle(values);
}

tria_distrib::tria_distrib(const params_list& params)
{
    _samples_type = sample_type::CONTINUOUS;

    if (!check_parameters(params))
        throw distribution_exception("Invalid parameters on
            ↪ triangular distribution.");

    _parameters = params;
}

base_distrib *tria_distrib::clone()
{
    return new tria_distrib(*this);
}

std::string tria_distrib::name() const
{
    return "Triangular";
}

std::vector<std::string> tria_distrib::paramNames() const
{
    std::vector<std::string> paramNames = std::vector<std::
        ↪ string>();
    paramNames.push_back("Min");
    paramNames.push_back("Mode");
    paramNames.push_back("Max");

    return paramNames;
}

data_type tria_distrib::get_mean() const
{
    return (_parameters[MIN] + _parameters[MODE] +
        ↪ _parameters[MAX]) / 3;
}

data_type tria_distrib::get_variance() const
```

```
{
    data_type a = _parameters[MIN];
    data_type c = _parameters[MODE];
    data_type b = _parameters[MAX];

    return ((a * a) + (b * b) + (c * c) - (a * b) - (a * c)
        ↪ - (b * c)) / 18;
}

data_type tria_distrib::get_mode() const
{
    return _parameters[MODE];
}

data_type tria_distrib::cumulative_function(data_type number
    ↪ ) const
{
    data_type result = 0.0;
    data_type a = _parameters[MIN];
    data_type c = _parameters[MODE];
    data_type b = _parameters[MAX];

    if (number < a)
        result = 0;
    else if ( (number >= a) && (number <= c) )
        result = ( (number - a) * (number - a) ) / ( (b - a)
            ↪  * (c - a) );
    else if ( (number > c) && (number <= b) )
        result = 1 - ( ( (b - number) * (b - number) ) / ( (
            ↪  b - a) * (b - c) ) );
    else
        result = 1;

    return result;
}

data_type tria_distrib::probability_function(data_type
    ↪ number) const
{
    data_type a = _parameters[MIN];
    data_type c = _parameters[MODE];
    data_type b = _parameters[MAX];

    if ( (number < a) || (number > b) )
        return 0;

    if (number <= c) //a <= number <= c
        return 2 * (number - a) / ( (b - a) * (c - a) );
    else //c < number <= b
        return 2 * (b - number) / ( (b - a) * (b - c) );
}
```

```
void tria_distrib::set_params_mle(const data_set& values)
{
    data_type mean = utils::_mean(values);
    data_type min = *std::min_element(values.begin(), values
        ↪ .end());
    data_type max = *std::max_element(values.begin(), values
        ↪ .end());

    _parameters[MIN] = min;
    _parameters[MAX] = max;

    /* This method is only a poor estimator, nevertheless
     * triangular distribution is a rough model.
     * Czuber has a better method although its dependent
     * of a frequency table */
    _parameters[MODE] = (3 * mean) - (min + max);

    if (!check_parameters(_parameters))
        throw distribution_exception("Invalid parameters on
            ↪ triangular distribution.");
}

data_type tria_distrib::get_random() const
{
    data_type result = 0.0;
    data_type r = random_number();
    data_type min = _parameters[MIN];
    data_type max = _parameters[MAX];
    data_type mode = _parameters[MODE];
    data_type dx = (mode - min) / (max - min);

    if(r > dx)
        result = max - sqrt((1 - r) * (max - mode) * (max -
            ↪ min));
    else
        result = min + sqrt(r * (mode - min) * (max - min));

    return result;
}

bool tria_distrib::check_parameters(const params_list&
    ↪ params) const
{
    return ((params.size() == PARAMS_NUMBER) &&
            (params.at(MIN) < params.at(MODE)) &&
            (params.at(MODE) < params.at(MAX)));
}
```

Listagem C.56: tria_distrib.h

```
//
    ↪ ----------------------------------------------------------------
```

```
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪

#ifndef TRIA_DISTRIB_H_
#define TRIA_DISTRIB_H_

/*
 * Triangular Distribution
 */

#include "base_distrib.h"

namespace fit
{
class tria_distrib : public base_distrib
{
public:
    enum _tria_params
    {
        MIN,
        MODE,
        MAX,
        PARAMS_NUMBER
```

```
    };

    tria_distrib(const data_set& values); //gets
        ↪ distribution parameters from MLE
    tria_distrib(const params_list& params); //gets
        ↪ distribution parameters from params.

    //Return a clone of de distribuition allocated on heap
    base_distrib *clone();

    std::string name() const;
    std::vector<std::string> paramNames() const;

    ~tria_distrib() {};

    data_type get_mean() const;
    data_type get_variance() const;
    data_type get_mode() const;

    data_type cumulative_function(data_type number) const;
        ↪ //cumulative distribution function
    data_type probability_function(data_type number) const;
        ↪ //probability density function

    /* Sets distribution parameters through maximum
        ↪ likelihood procedures */
    void set_params_mle(const data_set& values);

    /* Gets a random value according to distribution */
    data_type get_random() const;

    bool check_parameters(const params_list& params) const;
};
}

#endif /*TRIA_DISTRIB_H_*/
```

Listagem C.57: unif_distrib.cpp

```
//
    ↪ ------------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
```

```
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪

#include <distribution/unif_distrib.h>
#include <algorithm>

using namespace fit;

unif_distrib::unif_distrib(const data_set& values)
{
    _samples_type = sample_type::CONTINUOUS;

    _parameters.resize(PARAMS_NUMBER);
    set_params_mle(values);
}

unif_distrib::unif_distrib(const params_list& params)
{
    _samples_type = sample_type::CONTINUOUS;

    if (!check_parameters(params))
        throw distribution_exception("Invalid parameters on
            ↪ uniform distribution.");

    _parameters = params;
}

base_distrib *unif_distrib::clone()
{
    return new unif_distrib(*this);
}

std::string unif_distrib::name() const
{
```

```
    return "Uniform";
}

std::vector<std::string> unif_distrib::paramNames() const
{
    std::vector<std::string> paramNames = std::vector<std::
        ↪ string>();
    paramNames.push_back("Min");
    paramNames.push_back("Max");

    return paramNames;
}

data_type unif_distrib::get_mean() const
{
    return (_parameters[MIN] + _parameters[MAX]) / 2.0;
}

data_type unif_distrib::get_variance() const
{
    data_type a = _parameters[MIN];
    data_type b = _parameters[MAX];

    return ((b - a) * (b - a)) / 12;
}

data_type unif_distrib::cumulative_function(data_type number
    ↪ ) const
{
    data_type result = 0.0;
    data_type a = _parameters[MIN];
    data_type b = _parameters[MAX];

    if (number < a)
        result = 0.0;
    else if (number > b)
        result = 1.0;
    else
        result = (number - a) / (b - a);

    return result;
}

data_type unif_distrib::probability_function(data_type
    ↪ number) const
{
    data_type a = _parameters[MIN];
    data_type b = _parameters[MAX];

    if ( (number < a) || (number > b) )
        return 0;
```

```
    return 1 / (b - a);
}

void unif_distrib::set_params_mle(const data_set& values)
{
    _parameters[MIN] = *std::min_element(values.begin(),
        ↪ values.end());
    _parameters[MAX] = *std::max_element(values.begin(),
        ↪ values.end());

    if (!check_parameters(_parameters))
        throw distribution_exception("Invalid parameters on
            ↪ uniform distribution.");
}

data_type unif_distrib::get_random() const
{
    data_type a = _parameters[MIN];
    data_type b = _parameters[MAX];
    return a + ((b - a) * random_number());
}

bool unif_distrib::check_parameters(const params_list&
    ↪ params) const
{
    return ((params.size() == PARAMS_NUMBER) && (params.at(
        ↪ MAX) > params.at(MIN)));
}
```

Listagem C.58: unif_distrib.h

```
//
    ↪ ------------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
```

```
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪

#ifndef UNIF_DISTRIB_H_
#define UNIF_DISTRIB_H_

/*
 * Uniform Distribution
 */

#include "base_distrib.h"

namespace fit
{
class unif_distrib : public base_distrib
{
public:
    enum _unif_params
    {
        MIN,
        MAX,
        PARAMS_NUMBER
    };

    unif_distrib(const data_set& values); //gets
        ↪ distribution parameters from MLE
    unif_distrib(const params_list& params); //gets
        ↪ distribution parameters from params.

    //Return a clone of de distribuition allocated on heap
    base_distrib *clone();

    std::string name() const;
    std::vector<std::string> paramNames() const;

    ~unif_distrib() {};

    data_type get_mean() const;
    data_type get_variance() const;

    data_type cumulative_function(data_type number) const;
```

```
        ↪ //cumulative distribution function
    data_type probability_function(data_type number) const;
        ↪ //probability density function

    /* Sets distribution parameters through maximum
        ↪ likelihood procedures */
    void set_params_mle(const data_set& values);

    /* Gets a random value according to distribution */
    data_type get_random() const;

    bool check_parameters(const params_list& params) const;
};
}

#endif /*UNIF_DISTRIB_H_*/
```

Listagem C.59: utils.cpp

```
#include "utils/utils.h"

#include <QMultiMap>
#include "cmath"

Utils::Utils()
{
}

// check if samples are discrete
bool Utils::checkDiscreteSamples(QList<double> samples)
{
    int i = 0;
    double intpart;
    while(i < samples.size() && std::modf(samples.at(i), &
        ↪ intpart) == 0.0)
        i++;

    return i == samples.size();
}

// scale given samples and get their scale factor and offset
QPair<QPair<double, double>, QList<double> > Utils::
    ↪ scaleSamples(fit::_distribution_type distrib_type,
    ↪ QList<double> samples)
{
    QPair<QPair<double, double>, QList<double> > result;
    QPair<double, double> transformation;
    QList<double> scaled_samples;
    double off_set;
    double scale_factor;

    // assuming samples are already ordered
```

```
double min_old = samples.at(0);
double max_old = samples.last();

if(distrib_type == fit::BETA) // range: [0,1] -> need to
    ↪ scale and offset
{
    if(min_old >= 0 && max_old <= 1)
    {
        // samples are already in range
        result.first = QPair<double, double>(0,1);
        result.second = samples;
    }else
    {
        double min = 0;
        double max = 1;

        off_set = -(((max - min)*min_old)/(max_old -
            ↪ min_old)) + min;
        scale_factor = (max - min)/(max_old - min_old);

        transformation = getInverseScaleAndOffSet(min,
            ↪ max, min_old, max_old);

        scaled_samples.push_back(min);
        for(int i = 1; i < samples.size()-1; i++)
            scaled_samples.push_back(scale_factor*
                ↪ samples.at(i) + off_set);
        scaled_samples.push_back(max);

        result.first = transformation;
        result.second = scaled_samples;
    }

}else if(distrib_type == fit::EXPONENTIAL ||
    ↪ distrib_type == fit::GAMMA|| // range: [0,inf) ->
    ↪ need to offset
     distrib_type == fit::WEIBULL || distrib_type ==
         ↪ fit::LOGNORMAL||
     distrib_type == fit::BETA || distrib_type ==
         ↪ fit::POISSON)
{
    if(min_old >=0)
    {
        // samples are already in range
        result.first = QPair<double, double>(0,1);
        result.second = samples;
    }
    else
    {
        off_set = -min_old;
        for(int i = 0; i < samples.size(); i++)
            scaled_samples.push_back(samples.at(i) +
```

```
                    ↪ off_set);
                transformation.first = -off_set;
                transformation.second = 1;

                result.first = transformation;
                result.second = scaled_samples;
            }
        }else
        {
            result.first = QPair<double, double>(0,1);
            result.second = samples;
        }
        return result;

}

QList<QColor> Utils::getSortedColors(int n)
{
    QList<QColor> colors;
    colors << Qt::red << Qt::green << Qt::blue << Qt::
        ↪ magenta << Qt::yellow << Qt::cyan
            << Qt::darkRed << Qt::darkGreen << Qt::darkBlue
                ↪ << Qt::darkYellow << Qt::darkCyan;

    QList<QColor> colors_result;
    for(int i = 0; i < n; i++)
    {
        colors_result.push_back(colors.at(i));
    }
    return colors_result;
}

QList<QPair<QString, QPair<int, double> > > Utils::
    ↪ convertHistToList(Histogram &histogram)
{
    QList<QPair<QString, QPair<int, double> > > hist_return;

    QList<HistogramBin> bins = histogram.getBins();
    int total_samples = histogram.getTotalOfSamples();

    int i;
    for(i = 0; i < bins.size(); i++)
    {
        HistogramBin bin = bins[i];
        QPair<int, double> frequency;
        QPair<QString, QPair<int, double> > bin_info;

        double percentage = bin.getFrequency() * 100 / (
            ↪ double)total_samples;
        frequency.first = bin.getFrequency();
        frequency.second = percentage;
```

```
        QString bin_range;
        if(i != bins.size()-1)
            bin_range = "[ " + QString::number(bin.getMin(),
                ↪  'g', 2) + ", " + QString::number(bin.
                ↪ getMax(), 'g', 2) + " )";
        else
            bin_range = "[ " + QString::number(bin.getMin(),
                ↪  'g', 2) + ", " + QString::number(bin.
                ↪ getMax(), 'g', 2) + " ]";

        bin_info.first = bin_range;
        bin_info.second = frequency;

        hist_return.push_back(bin_info);
    }

    return hist_return;
}

void Utils::copyFreqTableToTableWidget(fit::freq_table
    ↪ freq_table, QTableWidget *widget_table, int
    ↪ total_samples)
{
    widget_table->setRowCount(freq_table.size());
    widget_table->setColumnCount(2);
    widget_table->setHorizontalHeaderLabels(QString("
        ↪ Frequency;Percentage").split(";"));

    std::list<fit::freq_table_entry>::const_iterator
        ↪ iterator;
    int i = 0;
    for (iterator = freq_table.begin(); iterator !=
        ↪ freq_table.end(); ++iterator) {
        int frequency_int = (*iterator).get_entry_values().
            ↪ size();
        double percentage_double = frequency_int * 100 / (
            ↪ double)total_samples;
        QString min = QString::number((*iterator).
            ↪ get_min_value(), 'g', 2);
        QString max = QString::number((*iterator).
            ↪ get_max_value(), 'g', 2);
        QString frequency = QString::number(frequency_int);
        QString percentage = QString::number(
            ↪ percentage_double, 'g', 2) + "%";

        QTableWidgetItem *item_header, *item_freq, *
            ↪ item_percentage;
        if(i != (int)freq_table.size()-1)
            item_header = new QTableWidgetItem("[ " + min +
                ↪ ", " + max + " )");
        else
            item_header = new QTableWidgetItem("[ " + min +
```

```
                ↪ ", " + max + " ]");

        item_freq = new QTableWidgetItem(frequency);
        item_percentage = new QTableWidgetItem(percentage);

        item_freq->setFlags(item_freq->flags() ^ Qt::
            ↪ ItemIsEditable);
        item_percentage->setFlags(item_percentage->flags() ^
            ↪ Qt::ItemIsEditable);

        widget_table->setVerticalHeaderItem(i, item_header);
        widget_table->setItem(i, 0, item_freq);
        widget_table->setItem(i, 1, item_percentage);

        i++;
    }
}

void Utils::sortDistributionsByChiStat(QList<DistributionFit
    ↪ *> &distributions)
{
    QMultiMap<double, DistributionFit*> sorted_distributions
        ↪ ;
    foreach(DistributionFit *distrib, distributions)
    {
        sorted_distributions.insert(distrib->getChiSquare().
            ↪ first.chi_statistic, distrib);
    }

    distributions.clear();

    foreach(DistributionFit *distrib, sorted_distributions)
    {
        distributions.push_back(distrib);
    }
}


QPair<double, double> Utils::getInverseScaleAndOffSet(double
    ↪ min, double max, double min_old, double max_old)
{
    double off_set = -(((max_old - min_old)*min)/(max - min)
        ↪ ) + min_old;
    double scale_factor = (max_old - min_old)/(max - min);

    return QPair<double, double>(off_set, scale_factor);
}
```

Listagem C.60: utils.h

```
#ifndef UTILS_H
#define UTILS_H
```

```
#include <QList>
#include <QColor>
#include <QPair>
#include <QTableWidget>
#include "fit.h"

#include "model/histogram.h"
#include "model/distribution_fit.h"

class Utils
{
public:
    Utils();
    static bool checkDiscreteSamples(QList<double> samples);
    static QPair<QPair<double, double> ,QList<double> >
        ↪ scaleSamples(fit::_distribution_type distrib_type
        ↪ ,QList<double> samples);
    static QList<QColor> getSortedColors(int n);
    static QList<QPair<QString, QPair<int, double> > >
        ↪ convertHistToList(Histogram &histogram);
    static void copyFreqTableToTableWidget(fit::freq_table
        ↪ freq_table, QTableWidget *widget_table, int
        ↪ total_samples);
    static void sortDistributionsByChiStat(QList<
        ↪ DistributionFit*> &distributions);

private:
    static QPair<double, double> getInverseScaleAndOffSet(
        ↪ double min, double max, double min_old, double
        ↪ max_old);
};

#endif // UTILS_H
```

Listagem C.61: weib_distrib.cpp

```
//
    ↪ ------------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
```

```
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ------------------------------------------------------------------------
    ↪

#include <distribution/weib_distrib.h>
#include <common/fit_utils.hpp>
#include <cmath>

using namespace fit;
using std::pow;
using std::exp;
using std::log;

weib_distrib::weib_distrib(const data_set& values)
{
    _samples_type = sample_type::CONTINUOUS;

    _parameters.resize(PARAMS_NUMBER);
    set_params_mle(values);
}

weib_distrib::weib_distrib(const params_list& params)
{
    _samples_type = sample_type::CONTINUOUS;

    if (!check_parameters(params))
        throw distribution_exception("Invalid parameters on
            ↪ weibull distribution.");

    _parameters = params;
}

base_distrib *weib_distrib::clone()
{
    return new weib_distrib(*this);
}
```

```
std::string weib_distrib::name() const
{
    return "Weibull";
}

std::vector<std::string> weib_distrib::paramNames() const
{
    std::vector<std::string> paramNames = std::vector<std::
        ↪ string>();
    paramNames.push_back("Alpha");
    paramNames.push_back("Beta");

    return paramNames;
}


data_type weib_distrib::get_mean() const
{
    data_type alpha = _parameters[ALPHA];
    data_type gamm_ret = exp(utils::gammln(1 / alpha));
    return gamm_ret * (_parameters[BETA] / alpha);
}

data_type weib_distrib::get_variance() const
{
    data_type alpha = _parameters[ALPHA];
    data_type beta = _parameters[BETA];
    data_type gamm_a = exp(utils::gammln(1 / alpha));
    data_type gamm_b = exp(utils::gammln(2 / alpha));
    data_type sec_calc = (1 / alpha) * (gamm_a * gamm_a);

    return ((beta * beta) / alpha) * ((2 * gamm_b) -
        ↪ sec_calc);
}

data_type weib_distrib::get_mode() const
{
    data_type alpha = _parameters[ALPHA];

    if (alpha < 1)
        return 0;

    return _parameters[BETA] * pow((alpha - 1) / alpha, (1 /
        ↪  alpha));
}

data_type weib_distrib::cumulative_function(data_type number
    ↪ ) const
{
    if (number <= 0)
        return 0;
```

```
    return 1 - exp((-1) * pow( (number / _parameters[BETA]),
        ↪  _parameters[ALPHA]) );
}

data_type weib_distrib::probability_function(data_type
    ↪ number) const
{
    if (number <=0)
        return 0;

    data_type alpha = _parameters[ALPHA];
    data_type beta = _parameters[BETA];
    return pow(alpha * beta, -alpha) * pow(number, alpha -
        ↪ 1) * exp(-pow(number/beta, alpha));
}

void weib_distrib::set_params_mle(const data_set& values)
{
    if (!check_values(values))
        throw distribution_exception("Invalid value on
            ↪ weibull distribution.");

    using utils::PI;
    using utils::EPS;
    using utils::ITMAX;

    data_type values_num = values.size();

    data_set::const_iterator i;

    data_type temp = 0.0;
    data_type sumValuesln = 0.0;
    data_type sumValueslnSqr = 0.0;

    for (i = values.begin(); i != values.end(); i++)
    {
        temp = log(*i);
        sumValuesln += temp;
        sumValueslnSqr += (temp * temp);
    }

    data_type constA = sumValuesln / values_num;
    data_type valueB = 0.0;
    data_type valueC = 0.0;
    data_type valueH = 0.0;

    data_type old_alpha = 0.0;
    data_type new_alpha = 0.0;

    /*Estimativa do primeiro old_alpha*/
    old_alpha = sqrt((values_num-1)/((6/(PI*PI))*(
        ↪ sumValueslnSqr-((sumValuesln*sumValuesln)/
```

```
        ↪ values_num))));

    int j = 0;
    for (j = 1; j <= ITMAX; j++)
    {
        for (i = values.begin(); i != values.end(); i++)
        {
            data_type valueLOG = log(*i);
            data_type valuePOW = pow(*i,old_alpha);
            valueB += valuePOW;
            valueC += valuePOW * valueLOG;
            valueH += valuePOW * (valueLOG * valueLOG);
        }
        new_alpha = old_alpha + ((constA + (1/old_alpha) - (
            ↪ valueC/valueB)) / ((1/(old_alpha*old_alpha))
            ↪ + (((valueB*valueH) - (valueC*valueC))/(
            ↪ valueB*valueB))));

        if ((new_alpha - old_alpha) < EPS)
        {
            break;
        }
        else
        {
            valueB = 0.0;
            valueC = 0.0;
            valueH = 0.0;
            old_alpha = new_alpha;
        }
    }
    if (j > ITMAX)
        throw distribution_exception("It was not possible to
            ↪  estimate weibull's parameters");

    _parameters[ALPHA] = new_alpha;
    _parameters[BETA] = pow((valueB/values_num),(1/new_alpha
        ↪ ));

    if (!check_parameters(_parameters))
        throw distribution_exception("Invalid parameters on
            ↪ weibull distribution.");
}

data_type weib_distrib::get_random() const
{
    data_type alpha = _parameters[ALPHA];
    data_type beta = _parameters[BETA];

    return beta * pow(-log(random_number()), (1 / alpha)) ;
}

bool weib_distrib::check_parameters(const params_list&
```

```
    ↪ params) const
{
    return ((params.size() == PARAMS_NUMBER) &&
            (params.at(ALPHA) > 0) &&
            (params.at(BETA) > 0));
}
```

Listagem C.62: weib_distrib.h

```
//
    ↪ ---------------------------------------------------------------------
    ↪
//    'FIT', a library for fitting statistical distribution
//    Copyright (C) 2007 Sanjay Formighieri < sanjayfm at
    ↪ gmail dot com >
//
//    This library is free software; you can redistribute it
    ↪  and/or
//    modify it under the terms of the GNU Lesser General
    ↪ Public
//    License as published by the Free Software Foundation;
    ↪ either
//    version 2.1 of the License, or (at your option) any
    ↪ later version.
//
//    This library is distributed in the hope that it will
    ↪ be useful,
//    but WITHOUT ANY WARRANTY; without even the implied
    ↪ warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    ↪ See the GNU
//    Lesser General Public License for more details.
//
//    You should have received a copy of the GNU Lesser
    ↪ General Public
//    License along with this library; if not, write to the
    ↪ Free Software
//    Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    ↪ MA 02111-1307 USA
//
    ↪ ---------------------------------------------------------------------
    ↪

#ifndef WEIB_DISTRIB_H_
#define WEIB_DISTRIB_H_

/*
 * Weibull Distribution
 */

#include "base_distrib.h"
```

```
namespace fit
{
class weib_distrib : public base_distrib
{
protected:
    inline bool in_range(data_type value) const {return
        ↪ value >= 0; }

public:
    enum _weib_params
    {
        ALPHA,
        BETA,
        PARAMS_NUMBER
    };

    weib_distrib(const data_set& values); //gets
        ↪ distribution parameters from MLE
    weib_distrib(const params_list& params); //gets
        ↪ distribution parameters from params.

    //Return a clone of de distribuition allocated on heap
    base_distrib *clone();

    std::string name() const;
    std::vector<std::string> paramNames() const;

    ~weib_distrib() {};

    data_type get_mean() const;
    data_type get_variance() const;
    data_type get_mode() const;

    data_type cumulative_function(data_type number) const;
        ↪ //cumulative distribution function
    data_type probability_function(data_type number) const;
        ↪ //probability density function

    /* Sets distribution parameters through maximum
        ↪ likelihood procedures */
    void set_params_mle(const data_set& values);

    /* Gets a random value according to distribution */
    data_type get_random() const;

    bool check_parameters(const params_list& params) const;

};
}

#endif /*WEIB_DISTRIB_H_*/
```

Listagem C.63: widget_chi_results.cpp

```
#include "view/widget_chi_results.h"
#include "ui_widget_chi_results.h"

#include <QRadioButton>

#include "utils/utils.h"

WidgetChiResults::WidgetChiResults(QWidget *parent) :
    QWidget(parent),
    ui_(new Ui::WidgetChiResults)
{
    ui_->setupUi(this);

    connect(&radio_button_group_, SIGNAL(buttonClicked(int))
        ↪ , this, SLOT(radioButtonClicked(int)));
}

WidgetChiResults::~WidgetChiResults()
{
    delete ui_;
}

void WidgetChiResults::showChiResults(const QList<
    ↪ DistributionFit *> distributions_fit)
{
    freq_tables_widgets_.clear();
    removeWidgetsFromLayout(ui_->vertical_layout_freq_tables
        ↪ );

    // add results to results table
    ui_->table_widget_results->setRowCount(distributions_fit
        ↪ .size());
    int i = 0;
    foreach(DistributionFit *distrib_fit, distributions_fit)
    {
        fit::chisquare_test_result distrib = distrib_fit->
            ↪ getChiSquare().first;
        fit::freq_table table = distrib_fit->getChiSquare().
            ↪ second;

        ui_->table_widget_results->verticalHeader()->hide();

        QRadioButton *radio_btn = new QRadioButton(
            ↪ distrib_fit->getName());
        radio_button_group_.addButton(radio_btn, i);

        ui_->table_widget_results->setCellWidget(i, 0,
            ↪ radio_btn);

        ui_->table_widget_results->setItem(i, 1, new
```

```
            ↪ QTableWidgetItem(QString::number(distrib.
            ↪ chi_statistic, 'g', 5)));
        ui_->table_widget_results->setItem(i, 2, new
            ↪ QTableWidgetItem(QString::number(distrib.
            ↪ chi_p_value, 'g', 5)));
        ui_->table_widget_results->setItem(i, 3, new
            ↪ QTableWidgetItem(QString::number(table.
            ↪ get_square_error(), 'g', 5)));

        // create frequency tables
        QTableWidget *freq_table_widget = new QTableWidget(
            ↪ this);
        freq_table_widget->horizontalHeader()->
            ↪ setStretchLastSection(true);
        freq_table_widget->verticalHeader()->
            ↪ setStretchLastSection(true);
        Utils::copyFreqTableToTableWidget(table,
            ↪ freq_table_widget, distrib_fit->getSamples().
            ↪ size());
        ui_->vertical_layout_freq_tables->addWidget(
            ↪ freq_table_widget);
        freq_tables_widgets_.push_back(freq_table_widget);
        if(i != 0)
            freq_table_widget->hide();
        else
        {
            id_current_freq_table_ = 0;
            radio_btn->setChecked(true);
        }

        i++;
    }
    ui_->table_widget_results->resizeColumnsToContents();
}

QList<QTableWidget *> WidgetChiResults::getFreqTablesWidgets
    ↪ () const
{
    return freq_tables_widgets_;
}

void WidgetChiResults::setFreqTablesWidgets(const QList<
    ↪ QTableWidget *> &freq_tables_widgets)
{
    freq_tables_widgets_ = freq_tables_widgets;
}

int WidgetChiResults::getIdCurrentFreqTable() const
{
    return id_current_freq_table_;
}
```

```
void WidgetChiResults::setIdCurrentFreqTable(int
    ↪ id_current_freq_table)
{
    id_current_freq_table_ = id_current_freq_table;
}

void WidgetChiResults::removeWidgetsFromLayout(QLayout *
    ↪ layout)
{
    QLayoutItem* child;
    while(layout->count()!=0)
    {
        child = layout->takeAt(0);
        if(child->layout() != 0)
        {
            removeWidgetsFromLayout(child->layout());
        }
        else if(child->widget() != 0)
        {
            delete child->widget();
        }

        delete child;
    }
}

void WidgetChiResults::radioButtonClicked(int id)
{
    freq_tables_widgets_[id_current_freq_table_]->hide();
    freq_tables_widgets_[id]->show();
    id_current_freq_table_ = id;
}
```

Listagem C.64: widget_chi_results.h

```
#ifndef WIDGET_CHI_RESULTS_H
#define WIDGET_CHI_RESULTS_H

#include <QWidget>
#include <QList>
#include <QTableWidget>
#include <QButtonGroup>

#include "model/distribution_fit.h"

namespace Ui {
class WidgetChiResults;
}

class WidgetChiResults : public QWidget
{
    Q_OBJECT
```

```
public:
    explicit WidgetChiResults(QWidget *parent = 0);
    ~WidgetChiResults();

    void showChiResults(const QList<DistributionFit*>
        ↪ distributions_fit);

    // GETTERS
    QList<QTableWidget *> getFreqTablesWidgets() const;
    int getIdCurrentFreqTable() const;

    // SETTERS
    void setFreqTablesWidgets(const QList<QTableWidget *> &
        ↪ freq_tables_widgets);
    void setIdCurrentFreqTable(int id_current_freq_table);

private:
    Ui::WidgetChiResults *ui_;
    QList<QTableWidget*> freq_tables_widgets_;
    QButtonGroup radio_button_group_;
    int id_current_freq_table_;

    void removeWidgetsFromLayout(QLayout* layout);

private slots:
    void radioButtonClicked(int id);
};

#endif // WIDGET_CHI_RESULTS_H
```

Listagem C.65: widget_chi_results.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>WidgetChiResults</class>
 <widget class="QWidget" name="WidgetChiResults">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>589</width>
    <height>333</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>Form</string>
  </property>
  <layout class="QVBoxLayout" name="verticalLayout">
   <item>
    <widget class="QGroupBox" name="group_box">
     <property name="font">
```

```
 <font>
  <weight>50</weight>
  <bold>false</bold>
 </font>
</property>
<property name="title">
 <string>Chi-Square results</string>
</property>
<layout class="QHBoxLayout" name="horizontalLayout_2">
 <item>
  <layout class="QVBoxLayout" name="verticalLayout_2">
   <item>
    <layout class="QHBoxLayout" name="horizontalLayout"
        ↪ >
     <item>
      <widget class="QLabel" name="label_2">
       <property name="font">
        <font>
         <weight>75</weight>
         <bold>true</bold>
        </font>
       </property>
       <property name="text">
        <string>Ranked fitted distributions</string>
       </property>
      </widget>
     </item>
     <item>
      <spacer name="horizontalSpacer">
       <property name="orientation">
        <enum>Qt::Horizontal</enum>
       </property>
       <property name="sizeHint" stdset="0">
        <size>
         <width>40</width>
         <height>20</height>
        </size>
       </property>
      </spacer>
     </item>
    </layout>
   </item>
   <item>
    <widget class="QTableWidget" name="
        ↪ table_widget_results">
     <property name="editTriggers">
      <set>QAbstractItemView::NoEditTriggers</set>
     </property>
     <attribute name="
         ↪ horizontalHeaderStretchLastSection">
      <bool>true</bool>
     </attribute>
```

```
      <attribute name="verticalHeaderStretchLastSection"
          ↪ >
       <bool>true</bool>
      </attribute>
      <column>
       <property name="text">
        <string>Distribution</string>
       </property>
       <property name="textAlignment">
        <set>AlignCenter</set>
       </property>
      </column>
      <column>
       <property name="text">
        <string>Chi-square</string>
       </property>
      </column>
      <column>
       <property name="text">
        <string>p-value</string>
       </property>
      </column>
      <column>
       <property name="text">
        <string>Square error</string>
       </property>
      </column>
     </widget>
    </item>
   </layout>
  </item>
  <item>
   <spacer name="horizontalSpacer_2">
    <property name="orientation">
     <enum>Qt::Horizontal</enum>
    </property>
    <property name="sizeType">
     <enum>QSizePolicy::Preferred</enum>
    </property>
    <property name="sizeHint" stdset="0">
     <size>
      <width>10</width>
      <height>20</height>
     </size>
    </property>
   </spacer>
  </item>
  <item>
   <layout class="QVBoxLayout" name="vertical_layout_3"
       ↪ stretch="0,1">
    <item>
     <layout class="QHBoxLayout" name="horizontal_layout
```

```
                ↪ " stretch="0,1">
               <item>
                <widget class="QLabel" name="label">
                 <property name="font">
                  <font>
                   <weight>75</weight>
                   <bold>true</bold>
                  </font>
                 </property>
                 <property name="text">
                  <string>Frequency table of selected
                      ↪ distribution</string>
                 </property>
                </widget>
               </item>
               <item>
                <spacer name="horizontal_spacer">
                 <property name="orientation">
                  <enum>Qt::Horizontal</enum>
                 </property>
                 <property name="sizeHint" stdset="0">
                  <size>
                   <width>40</width>
                   <height>20</height>
                  </size>
                 </property>
                </spacer>
               </item>
              </layout>
             </item>
             <item>
              <layout class="QVBoxLayout" name="
                  ↪ vertical_layout_freq_tables"/>
             </item>
            </layout>
           </item>
          </layout>
         </widget>
        </item>
       </layout>
      </widget>
      <resources/>
      <connections/>
     </ui>
```

Listagem C.66: widget_distrib_to_fit.cpp

```
#include "view/widget_distrib_to_fit.h"
#include "ui_widget_distrib_to_fit.h"

#include "view/widget_selectable_distrib.h"
```

```
WidgetDistribToFit::WidgetDistribToFit(QWidget *parent) :
    QWidget(parent),
    ui_(new Ui::WidgetDistribToFit)
{
    ui_->setupUi(this);

    createAllCheckableDistributions();
}

WidgetDistribToFit::~WidgetDistribToFit()
{
    delete ui_;
}

QList<int> WidgetDistribToFit::getSelectedDistributions()
{
    QList<int> indexes;

    QModelIndexList items_indexes = ui_->list_widget->
        ↪ selectionModel()->selectedIndexes();

    foreach(QModelIndex index, items_indexes)
    {
        indexes.push_back(index.row());
    }

    return indexes;
}

int WidgetDistribToFit::getBinCriteria()
{
    if(ui_->radio_button_sqrt_bin->isChecked())
        return 1;
    return 0;
}

void WidgetDistribToFit::createAllCheckableDistributions()
{
    QString name;
    QString info;
    QString img_resource;

    // discrete
    name = "Discrete Uniform";
    info = "Used when the random numbers are integers and
        ↪ they have the same probability, "
           "such as throwing a dice and getting a particular
               ↪ number. "
           "Parameters are the integers min and max. min is
               ↪ location parameter and"
           " max - min is scale parameter.";
    img_resource = ":/selectable_distrib/disc_unif_graph.svg
```

```
    ↪ ";
createCheckableDistribution(name, info, img_resource);

name = "Poisson";
info = "Number of events that occur in an interval of
    ↪ time when the events are occuring at a constant
    ↪ rate."
       " It is widely used in simulation. Parameter is
           ↪ the mean lambda.";
img_resource = ":/selectable_distrib/poisson_graph.svg";
createCheckableDistribution(name, info, img_resource);

// continuous
name = "Beta";
info = "Usually used as a rough model, due its
    ↪ simplicity. But there are some applications, such
    ↪  as"
       " the proportion of defective items in a batch
           ↪ production. Parameters are the shape
           ↪ parameter"
       " alpha1 and the scale parameter alpha2.";
img_resource = ":/selectable_distrib/beta_graph.svg";
createCheckableDistribution(name, info, img_resource);

name = "Exponential";
info = "Used as a model for interarrival times with
    ↪ strong randomness. Some applications are "
       "interarrival times of customers to a system that
           ↪  occur at a constant rate, time to failure
           ↪  of a piece of equipment."
       " The only parameter is the scale paramter beta."
           ↪ ;
img_resource = ":/selectable_distrib/expo_graph.svg";
createCheckableDistribution(name, info, img_resource);

name = "Gamma";
info = "Usually seen as a general form of exponential
    ↪ distribution, can be used as models such as time
    ↪ to complete a specific task, or "
       "any model of physical quantities that take
           ↪ positive values. Parameters are the shape
           ↪ parameter alpha and scale parameter beta."
           ↪ ;
img_resource = ":/selectable_distrib/gamma_graph.svg";
createCheckableDistribution(name, info, img_resource);

name = "Lognormal";
info = "Usually used as a rough model in the absence of
    ↪ data, the lognormal can algo be used to model
    ↪ time to perform tasks, "
       "among other applictions. Parameters are mean and
           ↪  standard deviation.";
```

```
    img_resource = ":/selectable_distrib/lognormal.svg";
    createCheckableDistribution(name, info, img_resource);

    name = "Normal";
    info = "Informally called the bell curve, the normal
        ↪ distribution is widely used. There are countless
        ↪ applications, such as "
           " measurement mistakes, the impact point of bomb,
               ↪  etc. Parameters are mean and standard
               ↪ deviation, which are the location "
           "and scale parameters.";
    img_resource = ":/selectable_distrib/normal_graph.svg";
    createCheckableDistribution(name, info, img_resource);

    name = "Triangular";
    info = "Used as rough model in the absence of data.
        ↪ Parameters are the min, mode and max. min is the
        ↪ location parameter, max - min is a scale "
           "parameter, and mode is a shape parameter.";
    img_resource = ":/selectable_distrib/triang_graph.svg";
    createCheckableDistribution(name, info, img_resource);

    name = "Uniform";
    info = "Used when the only knowledge are the minimum and
        ↪  maximum values, or as a uniform random generator
        ↪  for other distributions. Parameters "
           "are min and max";
    img_resource = ":/selectable_distrib/uniform.svg";
    createCheckableDistribution(name, info, img_resource);

    name = "Weibull";
    info = "Distribution often used to model a random
        ↪ variable of a equipment, such as a critical
        ↪ element that leads to failure. Parameters are "
           "the shape parameter alpha and the scale
               ↪ parameter beta.";
    img_resource = ":/selectable_distrib/weibull.svg";
    createCheckableDistribution(name, info, img_resource);
}

void WidgetDistribToFit::createCheckableDistribution(const
    ↪ QString name, const QString info, const QString
    ↪ img_resource)
{
    WidgetSelectableDistrib *widget = new
        ↪ WidgetSelectableDistrib(name, info, img_resource,
        ↪  this);
    QListWidgetItem *item = new QListWidgetItem();
    item->setSizeHint(widget->size());
    ui_->list_widget->addItem(item);
    ui_->list_widget->setItemWidget(item,widget);
}
```

```
void WidgetDistribToFit::on_radio_button_discretes_clicked(
    ↪ bool checked)
{
    if(!checked)
        return;
    ui_->list_widget->item(0)->setSelected(true);
    ui_->list_widget->item(1)->setSelected(true);
    for(int i = 2; i < ui_->list_widget->count(); i++)
    {
        ui_->list_widget->item(i)->setSelected(false);
    }
}

void WidgetDistribToFit::on_radio_button_continuous_clicked(
    ↪ bool checked)
{
    if(!checked)
        return;
    ui_->list_widget->item(0)->setSelected(false);
    ui_->list_widget->item(1)->setSelected(false);
    for(int i = 2; i < ui_->list_widget->count(); i++)
    {
        ui_->list_widget->item(i)->setSelected(true);
    }
}

void WidgetDistribToFit::on_radio_button_all_clicked(bool
    ↪ checked)
{
    if(!checked)
        return;
    for(int i = 0; i < ui_->list_widget->count(); i++)
    {
        ui_->list_widget->item(i)->setSelected(true);
    }
}
```

Listagem C.67: widget_distrib_to_fit.h

```
#ifndef WIDGET_DISTRIB_TO_FIT_H
#define WIDGET_DISTRIB_TO_FIT_H

#include <QWidget>

namespace Ui {
class WidgetDistribToFit;
}

class WidgetDistribToFit : public QWidget
{
    Q_OBJECT
```

```
public:
    explicit WidgetDistribToFit(QWidget *parent = 0);
    ~WidgetDistribToFit();

    QList<int> getSelectedDistributions();
    int getBinCriteria();

private:
    Ui::WidgetDistribToFit *ui_;

    void createAllCheckableDistributions();
    void createCheckableDistribution(const QString name,
        ↪ const QString info, const QString img_resource);

private slots:
    void on_radio_button_discretes_clicked(bool checked);
    void on_radio_button_continuous_clicked(bool checked);
    void on_radio_button_all_clicked(bool checked);
};

#endif // WIDGET_DISTRIB_TO_FIT_H
```

Listagem C.68: widget_distrib_to_fit.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>WidgetDistribToFit</class>
 <widget class="QWidget" name="WidgetDistribToFit">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>468</width>
    <height>237</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>Form</string>
  </property>
  <layout class="QHBoxLayout" name="horizontalLayout">
   <item>
    <layout class="QVBoxLayout" name="verticalLayout_3"
        ↪ stretch="0,0,0,1">
     <item>
      <widget class="QGroupBox" name="groupBox">
       <property name="title">
        <string>Distributions to fit</string>
       </property>
       <layout class="QVBoxLayout" name="verticalLayout">
        <item>
```

```
      <widget class="QRadioButton" name="
          ↪ radio_button_choose">
       <property name="text">
        <string>Choose</string>
       </property>
       <property name="checked">
        <bool>true</bool>
       </property>
      </widget>
     </item>
     <item>
      <widget class="QRadioButton" name="
          ↪ radio_button_discretes">
       <property name="text">
        <string>All Discretes</string>
       </property>
      </widget>
     </item>
     <item>
      <widget class="QRadioButton" name="
          ↪ radio_button_continuous">
       <property name="text">
        <string>All Continuous</string>
       </property>
      </widget>
     </item>
     <item>
      <widget class="QRadioButton" name="radio_button_all
          ↪ ">
       <property name="text">
        <string>Select All</string>
       </property>
       <property name="checked">
        <bool>false</bool>
       </property>
      </widget>
     </item>
    </layout>
   </widget>
  </item>
  <item>
   <spacer name="verticalSpacer_2">
    <property name="orientation">
     <enum>Qt::Vertical</enum>
    </property>
    <property name="sizeType">
     <enum>QSizePolicy::Fixed</enum>
    </property>
    <property name="sizeHint" stdset="0">
     <size>
      <width>20</width>
      <height>10</height>
```

```
    </size>
   </property>
  </spacer>
 </item>
 <item>
  <widget class="QGroupBox" name="groupBox_2">
   <property name="title">
    <string>Chi-Square Test</string>
   </property>
   <layout class="QVBoxLayout" name="verticalLayout_2">
    <item>
     <widget class="QLabel" name="label_bin_criteria">
      <property name="font">
       <font>
        <weight>75</weight>
        <bold>true</bold>
       </font>
      </property>
      <property name="text">
       <string>Bin criteria</string>
      </property>
     </widget>
    </item>
    <item>
     <widget class="QRadioButton" name="
         ↪ radio_button_sqrt_bin">
      <property name="text">
       <string>sqrt(n)</string>
      </property>
      <property name="checked">
       <bool>true</bool>
      </property>
     </widget>
    </item>
    <item>
     <widget class="QRadioButton" name="
         ↪ radio_button_sturges_bin">
      <property name="text">
       <string>1.5+3.222 * nlog(n)</string>
      </property>
      <property name="checked">
       <bool>false</bool>
      </property>
     </widget>
    </item>
   </layout>
  </widget>
 </item>
 <item>
  <spacer name="verticalSpacer">
   <property name="orientation">
    <enum>Qt::Vertical</enum>
```

```
        </property>
        <property name="sizeHint" stdset="0">
         <size>
          <width>20</width>
          <height>40</height>
         </size>
        </property>
       </spacer>
      </item>
     </layout>
    </item>
    <item>
     <widget class="QListWidget" name="list_widget">
      <property name="selectionMode">
       <enum>QAbstractItemView::MultiSelection</enum>
      </property>
      <property name="resizeMode">
       <enum>QListView::Adjust</enum>
      </property>
      <property name="viewMode">
       <enum>QListView::ListMode</enum>
      </property>
      <property name="uniformItemSizes">
       <bool>true</bool>
      </property>
     </widget>
    </item>
   </layout>
  </widget>
  <resources/>
  <connections/>
</ui>
```

Listagem C.69: widget_distribution_parameters.cpp

```
#include "view/widget_distribution_parameters.h"
#include "ui_widget_distribution_parameters.h"

#include "fit_defs.h"

WidgetDistributionParameters::WidgetDistributionParameters(
    ↪ const QStringList param_names, QWidget *parent) :
    QWidget(parent),
    ui_(new Ui::WidgetDistributionParameters)
{
    ui_->setupUi(this);


    int i = 0;
    foreach (const QString &parameter, param_names)
    {
        QLineEdit *line_edit = new QLineEdit(this);
```

```
        line_edit->setFixedWidth(70);
        line_edit->setValidator(new QDoubleValidator(this));

        QLabel *label = new QLabel(parameter + ":", this);
        label->setFixedWidth(40);

        param_values_.push_back(QPair<QString,QLineEdit*>(
            ↪ parameter, line_edit));
        ui_->grid_layout->addWidget(label, i, 0);
        ui_->grid_layout->addWidget(param_values_.back().
            ↪ second, i, 1);
        i++;
    }
}

WidgetDistributionParameters::~WidgetDistributionParameters
    ↪ ()
{
    delete ui_;
}

QVector<double> WidgetDistributionParameters::getParams()
{
    QVector<double> params;
    for(int i = 0; i < param_values_.size(); i++)
    {
        if(param_values_.at(i).second->text().isEmpty())
            throw fit::value_exception("Enter a value for "
                ↪ + param_values_.at(i).first.toStdString()
                ↪  + ".");

        params.push_back(param_values_.at(i).second->text().
            ↪ toDouble());
    }
    return params;
}

QList<QPair<QString, QLineEdit *> >
    ↪ WidgetDistributionParameters::getParamValues() const
{
    return param_values_;
}

void WidgetDistributionParameters::setParamValues(const
    ↪ QList<QPair<QString, QLineEdit *> > &param_values)
{
    param_values_ = param_values;
}
```

Listagem C.70: widget_distribution_parameters.h

```
#ifndef WIDGET_DISTRIBUTION_PARAMETERS_H
```

```
#define WIDGET_DISTRIBUTION_PARAMETERS_H

#include <QWidget>
#include <QLineEdit>

namespace Ui {
class WidgetDistributionParameters;
}

class WidgetDistributionParameters : public QWidget
{
    Q_OBJECT

public:
    explicit WidgetDistributionParameters(const QStringList
        ↪ param_names, QWidget *parent = 0);
    ~WidgetDistributionParameters();
    QVector<double> getParams();

    // GETTERS
    QList<QPair<QString, QLineEdit *> > getParamValues()
        ↪ const;

    // SETTERS
    void setParamValues(const QList<QPair<QString, QLineEdit
        ↪  *> > &param_values);

private:
    Ui::WidgetDistributionParameters *ui_;
    QList<QPair<QString,QLineEdit*> > param_values_;
};

#endif // WIDGET_DISTRIBUTION_PARAMETERS_H
```

Listagem C.71: widget_distribution_parameters.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>WidgetDistributionParameters</class>
 <widget class="QWidget" name="WidgetDistributionParameters"
     ↪ >
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>73</width>
    <height>80</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>Form</string>
  </property>
```

```
  <layout class="QFormLayout" name="formLayout">
   <property name="leftMargin">
    <number>0</number>
   </property>
   <property name="topMargin">
    <number>0</number>
   </property>
   <property name="rightMargin">
    <number>0</number>
   </property>
   <property name="bottomMargin">
    <number>0</number>
   </property>
   <item row="0" column="0">
    <widget class="QLabel" name="label_parameters">
     <property name="text">
      <string>Parameters</string>
     </property>
    </widget>
   </item>
   <item row="1" column="0">
    <layout class="QGridLayout" name="grid_layout"/>
   </item>
   <item row="2" column="0">
    <spacer name="vertical_spacer">
     <property name="orientation">
      <enum>Qt::Vertical</enum>
     </property>
     <property name="sizeHint" stdset="0">
      <size>
       <width>20</width>
       <height>40</height>
      </size>
     </property>
    </spacer>
   </item>
  </layout>
 </widget>
 <resources/>
 <connections/>
</ui>
```

Listagem C.72: widget_rand_number_generator.cpp

```
#include "view/widget_rand_number_generator.h"
#include "ui_widget_rand_number_generator.h"

#include "fit_defs.h"

WidgetRandNumberGenerator::WidgetRandNumberGenerator(QWidget
    ↪ *parent) :
    QWidget(parent),
```

```
    ui_(new Ui::WidgetRandNumberGenerator)
{
    ui_->setupUi(this);

    ui_->line_edit_num_samples->setValidator(new
        ↪ QIntValidator(0, 100000, this));

    setUpDistribParams();
    widgets_params_.at(0)->show();
    ui_->combo_box_distribution->setCurrentIndex(0);
    current_widget_params_ = 0;
}

WidgetRandNumberGenerator::~WidgetRandNumberGenerator()
{
    delete ui_;
}

QVector<double> WidgetRandNumberGenerator::
    ↪ getDistributionParams()
{
    return widgets_params_.at(ui_->combo_box_distribution->
        ↪ currentIndex())->getParams();
}

int WidgetRandNumberGenerator::getNumSamples()
{
    QString value = ui_->line_edit_num_samples->text();
    if(value.isEmpty())
        throw fit::value_exception("Enter a value for number
            ↪  of samples.");

    return ui_->line_edit_num_samples->text().toInt();
}

int WidgetRandNumberGenerator::getDistribIndex()
{
    return ui_->combo_box_distribution->currentIndex();
}

void WidgetRandNumberGenerator::setUpDistribParams()
{
    QStringList params;

    // discrete uniform
    params << "Min" << "Max";
    createWidgetParams(params);


    // poisson
    params.clear();
    params << "Mean";
```

```
    createWidgetParams(params);

    // beta
    params.clear();
    params << "Alpha1" << "Alpha2";
    createWidgetParams(params);

    // exponential
    params.clear();
    params << "Beta";
    createWidgetParams(params);

    // gamma
    params.clear();
    params << "Alpha" << "Beta";
    createWidgetParams(params);

    // lognormal
    params.clear();
    params << "Mean" << "Std Dev";
    createWidgetParams(params);

    // normal
    params.clear();
    params << "Mean" << "Std Dev";
    createWidgetParams(params);

    // triangular
    params.clear();
    params << "Min" << "Mode" << "Max";
    createWidgetParams(params);

    // uniform
    params.clear();
    params << "Min" << "Max";
    createWidgetParams(params);

    // weibull
    params.clear();
    params << "Alpha" << "Beta";
    createWidgetParams(params);
}

void WidgetRandNumberGenerator::createWidgetParams(
    ↪ QStringList params)
{
    WidgetDistributionParameters *widget_params = new
        ↪ WidgetDistributionParameters(params);
    widgets_params_.push_back(widget_params);
    this->layout()->addWidget(widget_params);
    connect(ui_->combo_box_distribution, SIGNAL(
        ↪ currentIndexChanged(int)), this, SLOT(
```

```
        ↪ showWidgetParams(int)));
    widget_params->hide();
}

void WidgetRandNumberGenerator::showWidgetParams(int index)
{
    widgets_params_[current_widget_params_]->hide();
    current_widget_params_ = index;
    widgets_params_.at(index)->show();
}
```

Listagem C.73: widget_rand_number_generator.h

```
#ifndef WIDGET_RAND_NUMBER_GENERATOR_H
#define WIDGET_RAND_NUMBER_GENERATOR_H

#include <QWidget>
#include <QStringList>

#include "widget_distribution_parameters.h"

namespace Ui {
class WidgetRandNumberGenerator;
}

class WidgetRandNumberGenerator : public QWidget
{
    Q_OBJECT

public:
    explicit WidgetRandNumberGenerator(QWidget *parent = 0);
    ~WidgetRandNumberGenerator();

    // GETTERS
    QVector<double> getDistributionParams();
    int getNumSamples();
    int getDistribIndex();

private:
    Ui::WidgetRandNumberGenerator *ui_;
    QList<WidgetDistributionParameters*> widgets_params_;
    int current_widget_params_;

    void setUpDistribParams();
    void createWidgetParams(QStringList params);

// SLOTS
private slots:
    void showWidgetParams(int index);
};

#endif // WIDGET_RAND_NUMBER_GENERATOR_H
```

Listagem C.74: widget_rand_number_generator.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>WidgetRandNumberGenerator</class>
 <widget class="QWidget" name="WidgetRandNumberGenerator">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>130</width>
    <height>94</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>Form</string>
  </property>
  <layout class="QHBoxLayout" name="horizontalLayout">
   <item>
    <layout class="QFormLayout" name="form_layout_left">
     <item row="0" column="0">
      <widget class="QLabel" name="label_distribution">
       <property name="text">
        <string>Distribution</string>
       </property>
      </widget>
     </item>
     <item row="1" column="0" colspan="2">
      <widget class="QComboBox" name="combo_box_distribution
          ↪ ">
       <property name="sizePolicy">
        <sizepolicy hsizetype="Fixed" vsizetype="Fixed">
         <horstretch>0</horstretch>
         <verstretch>0</verstretch>
        </sizepolicy>
       </property>
       <item>
        <property name="text">
         <string>Discrete Uniform</string>
        </property>
       </item>
       <item>
        <property name="text">
         <string>Poisson</string>
        </property>
       </item>
       <item>
        <property name="text">
         <string>Beta</string>
        </property>
```

```
    </item>
    <item>
     <property name="text">
      <string>Exponential</string>
     </property>
    </item>
    <item>
     <property name="text">
      <string>Gamma</string>
     </property>
    </item>
    <item>
     <property name="text">
      <string>Lognormal</string>
     </property>
    </item>
    <item>
     <property name="text">
      <string>Normal</string>
     </property>
    </item>
    <item>
     <property name="text">
      <string>Triangular</string>
     </property>
    </item>
    <item>
     <property name="text">
      <string>Uniform</string>
     </property>
    </item>
    <item>
     <property name="text">
      <string>Weibull</string>
     </property>
    </item>
   </widget>
  </item>
  <item row="2" column="0">
   <widget class="QLabel" name="label_num_samples">
    <property name="text">
     <string>Number of samples</string>
    </property>
   </widget>
  </item>
  <item row="3" column="0">
   <widget class="QLineEdit" name="line_edit_num_samples"
       ↪ >
    <property name="sizePolicy">
     <sizepolicy hsizetype="Maximum" vsizetype="Fixed">
      <horstretch>0</horstretch>
      <verstretch>0</verstretch>
```

```
          </sizepolicy>
         </property>
         <property name="maximumSize">
          <size>
           <width>80</width>
           <height>16777215</height>
          </size>
         </property>
         <property name="text">
          <string>5000</string>
         </property>
        </widget>
       </item>
      </layout>
     </item>
     <item>
      <spacer name="horizontalSpacer">
       <property name="orientation">
        <enum>Qt::Horizontal</enum>
       </property>
       <property name="sizeType">
        <enum>QSizePolicy::Fixed</enum>
       </property>
       <property name="sizeHint" stdset="0">
        <size>
         <width>5</width>
         <height>20</height>
        </size>
       </property>
      </spacer>
     </item>
    </layout>
   </widget>
   <resources/>
   <connections/>
</ui>
```

Listagem C.75: widget_samples_and_histogram.cpp

```
#include "view/widget_samples_and_histogram.h"
#include "ui_widget_samples_and_histogram.h"

#include <QFileDialog>
#include <QSettings>
#include <QFileInfo>
#include <QMessageBox>
#include "fit_defs.h"

#include "utils/file_handler.h"
#include "defs.h"
```

```
WidgetSamplesAndHistogram::WidgetSamplesAndHistogram(QWidget
    ↪  *parent) :
    QWidget(parent),
    ui_(new Ui::WidgetSamplesAndHistogram)
{
    ui_->setupUi(this);

    ui_->widget_rand_generator->setDisabled(true);
    ui_->spin_box_bin->setValue(40);
    has_samples_ = false;
}

WidgetSamplesAndHistogram::~WidgetSamplesAndHistogram()
{
    delete ui_;
}

void WidgetSamplesAndHistogram::setSamples(QList<double>
    ↪ samples)
{
    emit sigSetSamples(samples);

    ui_->text_samples->clear();

    QString text = "";
    foreach (double sample, samples) {
        text += QString::number(sample) + "\n";
    }
    ui_->text_samples->insertPlainText(text);
}

QList<double> WidgetSamplesAndHistogram::getSamplesFromXlsx
    ↪ ()
{
    QList<double> samples;

    const QString DEFAULT_DIR_KEY("default_dir");
    QSettings my_settings;

    QString file = QFileDialog::getOpenFileName(this,
        tr("Open xlxs"),  my_settings.value(DEFAULT_DIR_KEY)
            ↪ .toString(), tr("Excel Files (*.xlsx)"));
    if (!file.isEmpty())
    {
        // saving last visited dir
        QFile file_aux(file);
        QFileInfo file_info(file_aux.fileName());
        QString path(file_info.path());
        my_settings.setValue(DEFAULT_DIR_KEY, path);

        // try to read file
        try
```

```
        {
            samples = FileHandler::readSamplesFromXlsx(file)
                ↪ ;
        }
        catch(...)
        {
            throw;
        }
    }
    return samples;
}

QList<double> WidgetSamplesAndHistogram::getSamplesFromTxt()
{
    QList<double> samples;

    const QString DEFAULT_DIR_KEY("default_dir");
    QSettings my_settings;

    QString file = QFileDialog::getOpenFileName(this,
        tr("Open text file"), my_settings.value(
            ↪ DEFAULT_DIR_KEY).toString(), tr("Text Files
            ↪ (*.txt)"));

    if (!file.isEmpty())
    {
        // saving last visited dir
        QFile file_aux(file);
        QFileInfo file_info(file_aux.fileName());
        QString path(file_info.path());
        my_settings.setValue(DEFAULT_DIR_KEY, path);

        // try to read file
        try
        {
            samples = FileHandler::readSamplesFromTxt(file);
        }
        catch(...)
        {
            throw;
        }
    }

    return samples;
}

void WidgetSamplesAndHistogram::generateRandNumb()
{
    // getting distribution info
    int num_samples;
    QVector<double> params;
    try
```

```
    {
        num_samples = ui_->widget_rand_generator->
            ↪ getNumSamples();
        params = ui_->widget_rand_generator->
            ↪ getDistributionParams();

        int distrib_index = ui_->widget_rand_generator->
            ↪ getDistribIndex();

        emit sigGenerateRandNumb(num_samples, params,
            ↪ distrib_index);
    }
    catch(...)
    {
        throw;
    }
}

void WidgetSamplesAndHistogram::createHistogram()
{
    if(ui_->radio_button_sturges_bin->isChecked())
        emit sigCreateHistogram(0);
    if(ui_->radio_button_sqrt_bin->isChecked())
        emit sigCreateHistogram(1);
    if(ui_->radio_button_manual_bin->isChecked())
    {
        int num_bins = ui_->spin_box_bin->value();
        emit sigCreateHistogram(2, num_bins);
    }
}

// SLOTS

void WidgetSamplesAndHistogram::
    ↪ on_button_confirm_samples_clicked()
{
    QList<double> samples;

    try
    {
        if(ui_->radio_button_xls->isChecked())
        {
            // get samples from xlsx
            samples = getSamplesFromXlsx();

            if(samples.isEmpty())
                return;
            setSamples(samples);

        }else if(ui_->radio_button_txt->isChecked())
        {
            // get samples from text file
```

```
            samples = getSamplesFromTxt();

            if(samples.isEmpty())
                return;
            setSamples(samples);

        }
        else
        {
            // generate random samples
            generateRandNumb();
        }
    }
    catch(file_exception e)
    {
        QMessageBox::critical(0,"Error",e.msg);

        return;
    }
    catch(fit::value_exception e)
    {
        QMessageBox::critical(0,"Error",QString::
            ↪ fromStdString(e.msg));

        return;
    }
    catch(fit::distribution_exception e)
    {
        QMessageBox::critical(0,"Error",QString::
            ↪ fromStdString(e.msg));

        return;
    }

    // create histogram and plot
    createHistogram();

    has_samples_ = true;
}

void WidgetSamplesAndHistogram::
    ↪ on_radio_button_sturges_bin_clicked(bool checked)
{
    if(checked && has_samples_)
    {
        emit sigCreateHistogram(0);
    }
}

void WidgetSamplesAndHistogram::
    ↪ on_radio_button_sqrt_bin_clicked(bool checked)
{
```

```
    if(checked && has_samples_)
    {
        emit sigCreateHistogram(1);
    }
}

void WidgetSamplesAndHistogram::
    ↪ on_radio_button_manual_bin_clicked(bool checked)
{
    if(checked && has_samples_)
    {
        int num_bins = ui_->spin_box_bin->value();
        emit sigCreateHistogram(2, num_bins);
    }

}

void WidgetSamplesAndHistogram::on_spin_box_bin_valueChanged
    ↪ (int num_bins)
{
    if(has_samples_)
    {
        emit sigCreateHistogram(2, num_bins);
    }
}
```

Listagem C.76: widget_samples_and_histogram.h

```
#ifndef WIDGET_SAMPLES_AND_HISTOGRAM_H
#define WIDGET_SAMPLES_AND_HISTOGRAM_H

#include <QWidget>

namespace Ui {
class WidgetSamplesAndHistogram;
}

class WidgetSamplesAndHistogram : public QWidget
{
    Q_OBJECT

public:
    explicit WidgetSamplesAndHistogram(QWidget *parent = 0);
    ~WidgetSamplesAndHistogram();

    void setSamples(QList<double> samples);

private:
    Ui::WidgetSamplesAndHistogram *ui_;
    bool has_samples_;

    QList<double> getSamplesFromXlsx();
```

```
    QList<double> getSamplesFromTxt();
    void generateRandNumb();
    void createHistogram();

private slots:
    void on_button_confirm_samples_clicked();
    void on_radio_button_sturges_bin_clicked(bool checked);
    void on_radio_button_sqrt_bin_clicked(bool checked);
    void on_radio_button_manual_bin_clicked(bool checked);
    void on_spin_box_bin_valueChanged(int num_bins);

signals:
    void sigSetSamples(QList<double> samples);
    void sigGenerateRandNumb(const int numb_samples, const
        ↪ QVector<double> params, const int distrib_type);
    void sigCreateHistogram(const int bin_criteria, const
        ↪ int num_bins = 0);
    void sigEnableNext(bool enable);
};

#endif // WIDGET_SAMPLES_AND_HISTOGRAM_H
```

Listagem C.77: widget_samples_and_histogram.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>WidgetSamplesAndHistogram</class>
 <widget class="QWidget" name="WidgetSamplesAndHistogram">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>462</width>
    <height>165</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>Form</string>
  </property>
  <layout class="QHBoxLayout" name="horizontalLayout">
   <item>
    <widget class="QGroupBox" name="groupBox">
     <property name="title">
      <string>Settings</string>
     </property>
     <layout class="QVBoxLayout" name="verticalLayout_4">
      <item>
       <layout class="QHBoxLayout" name="horizontalLayout_2"
           ↪ >
        <item>
         <layout class="QVBoxLayout" name="verticalLayout">
          <item>
```

```
 <widget class="QLabel" name="label_2">
  <property name="font">
   <font>
    <weight>75</weight>
    <bold>true</bold>
   </font>
  </property>
  <property name="text">
   <string>Getting samples</string>
  </property>
 </widget>
</item>
<item>
 <widget class="QRadioButton" name="
     ↪ radio_button_xls">
  <property name="text">
   <string>Import from xlsx file (from xlsx first
       ↪ column)</string>
  </property>
  <property name="checked">
   <bool>true</bool>
  </property>
  <attribute name="buttonGroup">
   <string notr="true">buttonGroup_2</string>
  </attribute>
 </widget>
</item>
<item>
 <widget class="QRadioButton" name="
     ↪ radio_button_txt">
  <property name="text">
   <string>Import from txt file</string>
  </property>
  <attribute name="buttonGroup">
   <string notr="true">buttonGroup_2</string>
  </attribute>
 </widget>
</item>
<item>
 <widget class="QRadioButton" name="
     ↪ radio_button_random">
  <property name="text">
   <string>Generate random samples</string>
  </property>
  <attribute name="buttonGroup">
   <string notr="true">buttonGroup_2</string>
  </attribute>
 </widget>
</item>
<item>
 <widget class="WidgetRandNumberGenerator" name="
     ↪ widget_rand_generator" native="true">
```

```
      <property name="enabled">
       <bool>false</bool>
      </property>
     </widget>
    </item>
    <item>
     <layout class="QHBoxLayout" name="
         ↪ horizontalLayout_5">
      <item>
       <widget class="QPushButton" name="
           ↪ button_confirm_samples">
        <property name="sizePolicy">
         <sizepolicy hsizetype="Maximum" vsizetype="
             ↪ Fixed">
          <horstretch>0</horstretch>
          <verstretch>0</verstretch>
         </sizepolicy>
        </property>
        <property name="text">
         <string>Confirm samples</string>
        </property>
       </widget>
      </item>
     </layout>
    </item>
   </layout>
  </item>
  <item>
   <layout class="QVBoxLayout" name="verticalLayout_2"
       ↪ >
    <item>
     <layout class="QHBoxLayout" name="
         ↪ horizontalLayout_4">
      <item>
       <widget class="Line" name="line">
        <property name="orientation">
         <enum>Qt::Vertical</enum>
        </property>
       </widget>
      </item>
      <item>
       <layout class="QVBoxLayout" name="
           ↪ verticalLayout_7">
        <item>
         <widget class="QLabel" name="
             ↪ label_bin_criteria">
          <property name="font">
           <font>
            <weight>75</weight>
            <bold>true</bold>
           </font>
          </property>
```

```
   <property name="text">
    <string>Bin criteria</string>
   </property>
  </widget>
 </item>
 <item>
  <widget class="QRadioButton" name="
      ↪ radio_button_sqrt_bin">
   <property name="text">
    <string>sqrt(n)</string>
   </property>
   <property name="checked">
    <bool>true</bool>
   </property>
   <attribute name="buttonGroup">
    <string notr="true">buttonGroup</string>
   </attribute>
  </widget>
 </item>
 <item>
  <widget class="QRadioButton" name="
      ↪ radio_button_sturges_bin">
   <property name="text">
    <string>1.5+3.222 * nlog(n)</string>
   </property>
   <property name="checked">
    <bool>false</bool>
   </property>
   <attribute name="buttonGroup">
    <string notr="true">buttonGroup</string>
   </attribute>
  </widget>
 </item>
 <item>
  <layout class="QHBoxLayout" name="
      ↪ horizontal_layout_manual_bin">
   <item>
    <widget class="QRadioButton" name="
        ↪ radio_button_manual_bin">
     <property name="text">
      <string>Define:</string>
     </property>
     <attribute name="buttonGroup">
      <string notr="true">buttonGroup</string>
     </attribute>
    </widget>
   </item>
   <item>
    <widget class="QSpinBox" name="spin_box_bin
        ↪ ">
     <property name="enabled">
      <bool>false</bool>
```

```
                    </property>
                    <property name="minimum">
                     <number>5</number>
                    </property>
                    <property name="maximum">
                     <number>40</number>
                    </property>
                   </widget>
                  </item>
                  <item>
                   <spacer name="horizontal_spacer_manual_bin"
                       ↪ >
                    <property name="orientation">
                     <enum>Qt::Horizontal</enum>
                    </property>
                    <property name="sizeHint" stdset="0">
                     <size>
                      <width>0</width>
                      <height>20</height>
                     </size>
                    </property>
                   </spacer>
                  </item>
                 </layout>
                </item>
                <item>
                 <spacer name="verticalSpacer">
                  <property name="orientation">
                   <enum>Qt::Vertical</enum>
                  </property>
                  <property name="sizeHint" stdset="0">
                   <size>
                    <width>20</width>
                    <height>40</height>
                   </size>
                  </property>
                 </spacer>
                </item>
                <item>
                 <layout class="QHBoxLayout" name="
                     ↪ horizontalLayout_3"/>
                </item>
               </layout>
              </item>
             </layout>
            </item>
           </layout>
          </item>
         </layout>
        </item>
       </layout>
      </item>
     </layout>
    </widget>
```

```
   </item>
   <item>
    <layout class="QVBoxLayout" name="verticalLayout_5">
     <item>
      <widget class="QLabel" name="label">
       <property name="text">
        <string>Samples</string>
       </property>
      </widget>
     </item>
     <item>
      <widget class="QPlainTextEdit" name="text_samples"/>
     </item>
    </layout>
   </item>
  </layout>
 </widget>
 <customwidgets>
  <customwidget>
   <class>WidgetRandNumberGenerator</class>
   <extends>QWidget</extends>
   <header>view/widget_rand_number_generator.h</header>
   <container>1</container>
  </customwidget>
 </customwidgets>
 <resources/>
 <connections>
  <connection>
   <sender>radio_button_xls</sender>
   <signal>clicked(bool)</signal>
   <receiver>widget_rand_generator</receiver>
   <slot>setDisabled(bool)</slot>
   <hints>
    <hint type="sourcelabel">
     <x>125</x>
     <y>67</y>
    </hint>
    <hint type="destinationlabel">
     <x>125</x>
     <y>135</y>
    </hint>
   </hints>
  </connection>
  <connection>
   <sender>radio_button_txt</sender>
   <signal>clicked(bool)</signal>
   <receiver>widget_rand_generator</receiver>
   <slot>setDisabled(bool)</slot>
   <hints>
    <hint type="sourcelabel">
     <x>125</x>
     <y>89</y>
```

```
    </hint>
    <hint type="destinationlabel">
     <x>125</x>
     <y>135</y>
    </hint>
   </hints>
  </connection>
  <connection>
   <sender>radio_button_random</sender>
   <signal>clicked(bool)</signal>
   <receiver>widget_rand_generator</receiver>
   <slot>setEnabled(bool)</slot>
   <hints>
    <hint type="sourcelabel">
     <x>125</x>
     <y>111</y>
    </hint>
    <hint type="destinationlabel">
     <x>125</x>
     <y>135</y>
    </hint>
   </hints>
  </connection>
  <connection>
   <sender>radio_button_manual_bin</sender>
   <signal>clicked(bool)</signal>
   <receiver>spin_box_bin</receiver>
   <slot>setEnabled(bool)</slot>
   <hints>
    <hint type="sourcelabel">
     <x>271</x>
     <y>104</y>
    </hint>
    <hint type="destinationlabel">
     <x>321</x>
     <y>105</y>
    </hint>
   </hints>
  </connection>
  <connection>
   <sender>radio_button_sturges_bin</sender>
   <signal>clicked(bool)</signal>
   <receiver>spin_box_bin</receiver>
   <slot>setDisabled(bool)</slot>
   <hints>
    <hint type="sourcelabel">
     <x>315</x>
     <y>58</y>
    </hint>
    <hint type="destinationlabel">
     <x>321</x>
     <y>105</y>
```

```
     </hint>
    </hints>
   </connection>
   <connection>
    <sender>radio_button_sqrt_bin</sender>
    <signal>clicked(bool)</signal>
    <receiver>spin_box_bin</receiver>
    <slot>setDisabled(bool)</slot>
    <hints>
     <hint type="sourcelabel">
      <x>315</x>
      <y>80</y>
     </hint>
     <hint type="destinationlabel">
      <x>321</x>
      <y>105</y>
     </hint>
    </hints>
   </connection>
  </connections>
  <buttongroups>
   <buttongroup name="buttonGroup"/>
   <buttongroup name="buttonGroup_2"/>
  </buttongroups>
</ui>
```

Listagem C.78: widget_selectable_distrib.cpp

```
#include "view/widget_selectable_distrib.h"
#include "ui_widget_selectable_distrib.h"

WidgetSelectableDistrib::WidgetSelectableDistrib(const
    ↪ QString name, const QString info,
                                                 const
                                                     ↪ QString
                                                     ↪
                                                     ↪ img_resource
                                                     ↪ ,
                                                     ↪ QWidget
                                                     ↪  *
                                                     ↪ parent
                                                     ↪ ) :
    QWidget(parent),
    ui_(new Ui::WidgetSelectableDistrib)
{
    ui_->setupUi(this);
    ui_->plain_text_edit->viewport()->unsetCursor();

    ui_->label_name_distrib->setText(name);
    ui_->plain_text_edit->insertPlainText(info);
    ui_->label_img->setPixmap(QPixmap(img_resource));
}
```

```
WidgetSelectableDistrib::~WidgetSelectableDistrib()
{
    delete ui_;
}
```

Listagem C.79: widget_selectable_distrib.h

```
#ifndef WIDGET_SELECTABLE_DISTRIB_H
#define WIDGET_SELECTABLE_DISTRIB_H

#include <QWidget>

namespace Ui {
class WidgetSelectableDistrib;
}

class WidgetSelectableDistrib : public QWidget
{
    Q_OBJECT

public:
    explicit WidgetSelectableDistrib(const QString name,
        ↪ const QString info, const QString img_resource,
        ↪ QWidget *parent = 0);
    ~WidgetSelectableDistrib();

private:
    Ui::WidgetSelectableDistrib *ui_;
};

#endif // WIDGET_SELECTABLE_DISTRIB_H
```

Listagem C.80: widget_selectable_distrib.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>WidgetSelectableDistrib</class>
 <widget class="QWidget" name="WidgetSelectableDistrib">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>200</width>
    <height>80</height>
   </rect>
  </property>
  <property name="sizePolicy">
   <sizepolicy hsizetype="Preferred" vsizetype="Fixed">
    <horstretch>0</horstretch>
    <verstretch>0</verstretch>
```

```
 </sizepolicy>
</property>
<property name="minimumSize">
 <size>
  <width>0</width>
  <height>80</height>
 </size>
</property>
<property name="windowTitle">
 <string>Form</string>
</property>
<layout class="QHBoxLayout" name="horizontalLayout_2"
    ↪ stretch="0,1">
 <item>
  <layout class="QVBoxLayout" name="vertical_layout"
      ↪ stretch="1,0">
   <item>
    <layout class="QHBoxLayout" name="horizontalLayout">
     <item>
      <widget class="QLabel" name="label_img">
       <property name="minimumSize">
        <size>
         <width>50</width>
         <height>0</height>
        </size>
       </property>
       <property name="maximumSize">
        <size>
         <width>80</width>
         <height>16777215</height>
        </size>
       </property>
       <property name="text">
        <string/>
       </property>
       <property name="scaledContents">
        <bool>true</bool>
       </property>
      </widget>
     </item>
    </layout>
   </item>
   <item>
    <layout class="QHBoxLayout" name="horizontal_layout">
     <item>
      <widget class="QLabel" name="label_name_distrib">
       <property name="minimumSize">
        <size>
         <width>100</width>
         <height>0</height>
        </size>
       </property>
```

```
        <property name="font">
         <font>
          <weight>75</weight>
          <bold>true</bold>
         </font>
        </property>
        <property name="text">
         <string>Name</string>
        </property>
        <property name="alignment">
         <set>Qt::AlignCenter</set>
        </property>
       </widget>
      </item>
     </layout>
    </item>
   </layout>
  </item>
  <item>
   <widget class="QPlainTextEdit" name="plain_text_edit">
    <property name="enabled">
     <bool>true</bool>
    </property>
    <property name="minimumSize">
     <size>
      <width>0</width>
      <height>0</height>
     </size>
    </property>
    <property name="styleSheet">
     <string notr="true"/>
    </property>
    <property name="frameShape">
     <enum>QFrame::NoFrame</enum>
    </property>
    <property name="sizeAdjustPolicy">
     <enum>QAbstractScrollArea::AdjustToContentsOnFirstShow
         ↪ </enum>
    </property>
    <property name="readOnly">
     <bool>true</bool>
    </property>
    <property name="plainText">
     <string/>
    </property>
    <property name="overwriteMode">
     <bool>false</bool>
    </property>
    <property name="cursorWidth">
     <number>0</number>
    </property>
    <property name="textInteractionFlags">
```

```
        <set>Qt::NoTextInteraction</set>
       </property>
      </widget>
     </item>
    </layout>
   </widget>
   <resources/>
   <connections/>
</ui>
```